UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ON-DEMAND LINK-STATE ROUTING IN AD-HOC NETWORKS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Soumya Roy

June 2003

The Dissertation of Soumya Roy
is approved:

Professor J.J. Garcia-Luna-Aceves, Chair

Professor Richard Hughey

Professor Katia Obraczka

Frank Talamantes
Vice Provost and Dean of Graduate Studies

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUN 2003		**00-06-2003 to 00-06-2003**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
On-Demand Link-State Routing in Ad-Hoc Networks	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**185**	

Contents

List of Figures

List of Tables

Abstract

On-Demand Link-State Routing in Ad-hoc Networks

by

Soumya Roy

This thesis explores the challenges, merits and demerits of using link-state information for on-demand routing in ad hoc networks, such that routers maintain path information for only those destinations for which they have data traffic.

We first present the source tree on-demand adaptive routing (SOAR) protocol, in which each router exchanges with its neighbors a "source tree" containing paths to only those destinations for which the router is the source or relay of data packets. The main advantage of SOAR is that it is more scalable and better performing than current state-of-the-art on-demand routing protocols. However, a limitation of SOAR is that it requires data packets to specify the paths they traverse to detect loops. To eliminate the need for source routing or path traversal information in data packets, we introduce the on-demand link-vector (OLIVE) protocol, which prevents temporary loops for each destination by synchronizing relevant link-state information among neighbors. In OLIVE, the advertised paths combine to form a source graph, rather than a source tree. OLIVE is shown to outperform the current routing protocols proposed for mobile ad-hoc networks in terms of control overhead, throughput and network delay.

We demonstrate that the problem of computing a source tree with the con-

straint imposed due to the exact nature of on-demand routing protocols is an NP-complete problem and show approximation algorithms for the path-selection problem.

The practical implementations of ad-hoc networks would be mainly wireless extensions of the wired Internet and the traffic would be mainly from the mobile nodes towards certain special nodes that act as gateways to the wired Internet. To achieve high performance in such scenarios we have developed a new genre of node-centric hybrid routing protocol where routes for frequently-accessed gateways would be kept proactive, while routes between mobile nodes would be reactive.

To My Parents and Brother

Acknowledgements

I am very grateful to my parents who have been my greatest guides all through-out my PhD life. This research and learning would never have been possible without the continuous support, unconditional love, affection and sacrifice of my parents. I give my deepest thanks to my elder brother, who has been my greatest well-wisher all the time and has always given me affection and support.

I am lucky to have Prof. JJ Garcia-Luna-Aceves as my adviser. He has never failed to give me the motivation and correct direction and I cannot express how greatly I benefited from his words of wisdom whenever I am facing problems.

I would like to thank Prof. Katia Obraczka and Prof. Richard Hughey for being in my advancement and defense committees. I also extend my thanks to Prof. Suresh Lodha who had been in my advancement committee and had shown considerable support for my work. Thanks also to Carol Mullane and Jodi Reiger, who had always been of great help to me.

I cannot forget the wonderful support of my friends, Architadi, Smita, Avik, Sasmal, Srikumar, Vidhya, Manju, Sanjit, Shailaja, Chandramouli, Joyopriya, Hemanth, Jyoti, Aman, Rahul, Rini, Rahul (Kundu), Raman, Subhajit, Subhasreedi, Manosizda, Amyn, Preethy, Deepa, Vineet, Bhagyashri, Shantanu, Ajoy, Vaibhav and Amin who have helped me gain confidence and happiness during the trying times and have made my school-life enjoyable. I am also grateful to my coco-inmates, Srinivas, Marcelo (Spohn), Marco, Marcelo (Carvalho), Marc, Lichun, and Yu for the wonderful research discussions I had with them and also to Ramesh, Saro, Hari, Ravindra and

Venkatesh for the challenging non-research discussions.

This work was supported in part by the Defense Research Projects Agency (DARPA) under grant F30602-97-2-0338 and by US Air Force/OSR under grant F49620-00-1-0330.

Chapter 1

Introduction

In today's Internet, wireless networks are becoming more prevalent as they can make access "anytime, anywhere" possible. Two main architectures for wireless networks are : wireless local access networks (WLANs), and wireless ad-hoc networks. The mobile nodes in a WLAN directly communicate with the fixed base-station to send their traffic to nodes in the same or different WLAN or to the nodes in the wired Internet. Routing is not an important issue in a WLAN because the path to the base station is one hop. However, routing is critical for data forwarding in ad-hoc networks, where each mobile node can act as a relay in addition to being a source or a destination of data packets. Ad-hoc networks can play an important role in relief scenarios and battlefields, which cannot count on a base-station infrastructure. The participants of conference scenarios or lecture sessions can also form an ad-hoc network and this network can get connected to the wired Internet through gateways.

1.1 On-demand Routing for Ad-Hoc Networks

Certain specific features like interference, fading, shadowing, low available bandwidth of wireless medium and mobility of nodes pose interesting challenges for developing routing solutions for wireless ad-hoc networks. The proactive routing protocols that have been designed for the wired Internet to achieve reliability, robustness, and optimality cannot be directly applied to wireless ad-hoc networks. For wireless ad-hoc networks the routing protocols can be proactive as well as reactive. Pro-active maintenance of paths to each node in an ad-hoc network can be expensive in terms of routing overhead because routes break and build frequently [9]. Moreover, continuous route maintenance at each node for every other destination may not be required, because all nodes need not communicate with every other node in the network. On-demand or reactive routing protocols, (e.g., DSR [25], AODV [33], TORA [32], ROAM [36], DST [35], NSR [44]) reduce the control overhead by maintaining paths to only those destinations to which data must be sent and the paths to such destinations need not be optimum. Given that the links in the network are not reliable, it is more important to set up and maintain at least one path, rather than requiring the optimal path. On-demand routing protocols can be different from one another based on how they communicate information to obtain paths to destinations, how they use and maintain that information, and the way in which data packets are routed. All on-demand routing protocols proposed to-date use flood search messages that either give sources the complete paths to destinations (e.g., DSR), or provide only the distances and next-hops to destinations and validate such distances in a number of ways (e.g.,

sequence numbers as in AODV, timestamps as in TORA, or internodal coordination as in ROAM).

Garcia-Luna-Aceves and Spohn [24] introduced the source-tree adaptive routing (STAR) protocol, in which a router informs its neighbors only the state of those links along the paths it chooses to reach all the known destinations in the ad-hoc network. The set of those links constitutes the source tree of a router. STAR has been shown to have very competitive performance compared to DSR and AODV [24, 23] while maintaining routing information for all network destinations. Link-state routing protocols like OSPF, used in today's Internet [28], are not suitable for mobile ad-hoc networks, because they depend on flooding of link-state information and replicating the entire topology information at each node of the network. Jacquet et al. [22] present a proactive link-state routing protocol for dense mobile ad-hoc networks called optimized link-state routing (OLSR). In OLSR, routers exchange periodic routing messages regarding all destinations, and periodic hello messages with neighbors. OLSR uses the concept of multipoint relays (MPRs), which act as intermediate routers from sources to destinations for forwarding of control information and works best in dense networks. Topology Dissemination based on Reverse Path Forwarding (TBRPF) [30] is another proactive link-state routing protocol, in which each router computes a source tree, consisting of paths to all destinations based on the partial topology information stored and reports part of the source tree to neighbors. The set of nodes in the reported part of the source tree is determined at any router based on whether, with respect to the destinations, it can act as relay for other neighbors (the

method is similar to selection of MPRs for OLSR [22]). TBRPF has been shown to perform better than AODV and OLSR in terms of optimality of paths, throughput and packet delay [29].

Given that each of the above link-state protocols is proactive and the source tree-based approaches of STAR and TBRPF show comparable performance to on-demand routing protocols, the obvious question is how to use source-tree information in an on-demand routing protocol. Therefore, the motivation for such a design is the desirability to provide a solution that is even more efficient than the current source-tree based proactive routing protocols or the distance vector or path based on-demand routing solutions by exploiting the use of on-demand link-state information. This leads to the first part of our research, namely designing and developing a protocol to use source-tree information on-demand, which we call the source tree on-demand adaptive routing (SOAR) protocol. Chapter 2 describes SOAR and compares it with AODV and DSR.

1.2 Path Selection Algorithm

Path selection algorithms like the Dijkstra's algorithm or the Bellman-Ford algorithm [6] are commonly used for computation of routes in a link-state routing protocol, in which the criterion for choosing the shortest path is a single metric. For on-demand link-state based routing protocols, these algorithms cannot be directly applied for path-selection because the criterion for the choice of successor is determined both by shortest path metric and by the set of neighbors who have advertised that route.

Because all nodes do not have paths for every destination, a situation can arise in which a node i finds that neighbor k should be on the shortest path to a destination j, but node k has not advertised any route to node j. Therefore, although node i knows that there exists a path to node j through node k, it cannot forward packets for node j to node k, because node k does not know how to reach destination j. To prevent packet losses, the path selection algorithm should ensure that the links in the anticipated path from node i to node j through node k have been advertised by node k itself. Given this constraint, traditional path selection algorithms like the Dijkstra's algorithm or the Bellman-Ford algorithm cannot find correctly shortest paths for all destinations. We have shown that building an optimal source tree i.e., the source tree with maximal number of vertices in it taking into consideration the above constraint, is an NP-complete problem. Therefore, we have developed a heuristic that computes the required source tree. Finite cost paths for some destinations will not exist in the resultant source tree. However, finite cost routes for the same destinations is possible if a source graph, rather than a source tree is computed. Chapter 3 describes the challenges of designing a path selection algorithm for ad-hoc networks with constraints because of the use of link-state information on-demand. Chapter 3 also describes how the proposed path selection algorithm can be extended to support loop-free policy-based routing for the Internet.

1.3 Instantaneous Loop Free Routing

Maintaining loop-free routes at every instant becomes extremely important in ad-hoc networks with dynamic topologies, because routing loops increase packet-delivery latencies and reduce the number of packets delivered to the intended destinations. Current on-demand routing protocols adopt different techniques to prevent temporary loops.

The dynamic source routing (DSR) protocol [25] is an example of protocols that attain loop-free routing using source routes. In DSR, each route-request, broadcast to find a destination, records its traversed route, and a route reply sent by a node specifies the complete route between the node and the destination. Routers store the discovered routes in a route cache. The header of every data packet in the basic scheme specifies the source routes to their intended destinations. However, implicit source routing has been proposed recently [20], in which data packets will contain flow identifiers and need not specify complete source routes. Implicit source routing renders only small increases in data packet overhead and impacts packet delivery ratios and average delays only slightly with respect to the basic DSR scheme.

The ad-hoc on-demand distance vector (AODV) protocol [34] is an example of maintaining loop-free routes by using a sequence number for each destination. In AODV, active routes for a given destination have sequence numbers that are non-increasing moving away from the destination. When a node A needs to establish a route to a destination D, it broadcasts a route request to its neighbors. If A previously knew a route to D that became invalid, A increases the sequence number for D and

includes it in the route request. A node receiving the request can send back a unicast route reply along its shortest path to node A only if it has a valid route to D and the sequence number stored for D is not smaller than the sequence number in the route request. Otherwise, the node receiving the route request must forward the route request. When node A sends a route request for a destination, it increases the sequence number for itself as well, which is used by other nodes that learn about new routes to node A. Increasing the sequence number for a destination when routes must be changed ensures loop freedom, but prevents nodes with valid and shorter paths to the destination from being used, and in many cases makes the destination the only node that can answer the route requests, i.e., forces unnecessary network-wide flooding.

The temporally-ordered routing algorithm (TORA) [32] uses a link-reversal algorithm [12] to maintain loop-free multipaths that are created by a query-reply process similar to that used in DSR and AODV. TORA relies on synchronized clocks to create timestamps that maintain the relative ordering of events. The link-reversal algorithm is a form of synchronization among nodes spanning multiple hops.

The routing on-demand acyclic multipath (ROAM) protocol [36] uses a similar approach in that it requires synchronization of the nodes' routing-table updates across multiple hops. Although these approaches provide loop-free routing and permit local repair of routes, their main disadvantage is that they require reliable exchanges among neighbors and coordination among nodes over multiple hops, thus incurring more control messages compared to AODV, DSR, and other on-demand protocols that work correctly even with unreliable transmissions of route requests and replies

among neighbors.

The source tree on-demand adaptive routing (SOAR) protocol that we have mentioned before, requires data packets to carry the path traversal information in order to detect routing loops.

In Chapter 4 we present the on-demand link vector protocol (OLIVE), which is the first on-demand routing protocol based on path information specified as vectors of link-states and supports loop-free incremental routing based simply on the destination of data packets. OLIVE does not need internodal synchronization spanning multiple hops, the use of a source route, a path traversed, or a flow identifier in the header of data packets or the use of destination-based sequence numbers. OLIVE generalizes the exchange of path information attained in DSR and other protocols (e.g., the neighborhood-aware source routing protocol (NSR) [44]) such that, when a router finds that selecting a route for a destination after a network change can lead to a potential loop, it forces all its neighbors to tell the router when they stop using it in their own paths to the same destination. In so doing the neighbors of the router provide the router with zero or some alternate loop-free paths to the destination. This process requires coordination between immediate one-hop neighbors only.

1.4 Node-Centric Hybrid Routing

Prior simulation studies [38, 9, 8] to test the performance of on-demand routing protocols in mobile ad-hoc networks have assumed uniform traffic pattern in the network, i.e., traffic flows are randomly distributed throughout the network, with

no node being accessed more than others. However, in practical scenarios of ad-hoc networks, traffic patterns can be non-uniformly distributed and concentrate around nodes that host popular network services (e.g., DNS services, Internet access, web proxies) that are requested throughout the ad-hoc network. When only a few nodes of the ad-hoc network must act as sources and sinks of most of the data packets, maintaining routing information to such nodes on-demand and treating those nodes as any other node may not be as attractive as a proactive approach for establishing routing information to them while on-demand routing is used between less accessed nodes. This motivates the interest in a node-centric hybrid approach to routing in ad-hoc networks.

Chapter 5 introduces two approaches to node-centric hybrid routing. In one approach, the netmark, which is the heavily accessed node, forces the rest of the nodes to maintain their routes to it for long periods of time once they acquire it. This amounts to extending the caching of netmark routing information. In another approach, a netmark uses proactive routing updates to push its routing entry into the routing tables of the rest of the nodes in the ad-hoc network.

Chapter 2

Source Tree On Demand Adaptive Routing (SOAR)

The source-tree on-demand adaptive routing (SOAR) protocol is an on-demand link-state routing protocol designed for wireless ad-hoc networks. The key idea in SOAR is that the wireless routers exchange *minimal source trees*, consisting of the state of the links that are in the paths used by the routers to reach *important* destinations. Important destinations are active receivers, relays, or future potential relays of data packets. Minimal source trees can be updated incrementally or atomically, and updates to individual links in source trees are validated using sequence numbers. A wireless router uses its outgoing links and the minimal source trees received from its neighbors to get a partial view of the network and computes its source tree by running a local path selection algorithm on its partial topology.

Like any other on-demand routing protocol, SOAR finds paths to destinations

in an on-demand basis using *queries* and *replies*. When a router receives a data packet to forward and the router already has an entry for the intended destination in its routing table, it forwards the packet to the next hop specified in its routing table. If a router has no route for the destination of a data packet, the router sends a query to its neighbors asking for the link-state information needed to produce a complete path to the destination. A router that receives a query and does not have a path for the requested destination forwards the query to its own neighbors; otherwise, it responds with a reply. Routers exchange *updates* during network connectivity changes to prevent loops and incorrect packet forwarding.

Section 2.1 describes in detail how SOAR works. Section 2.2 describes the correctness of SOAR. Section 2.3 compares the performance of SOAR with AODV and DSR, two popular on-demand routing protocols proposed for mobile ad-hoc networks. Section 2.4 concludes the chapter.

2.1 Working of SOAR

To describe SOAR, we model the topology of an ad-hoc network as a directed graph $G = (\ V, E\ )$, where V is the set of nodes and E is the set of edges connecting the nodes. When a router finds a new node in its range, it assumes the existence of a new link with it. Every link can have a cost associated with it, which becomes infinite when the neighbor is not directly reachable. Each node has a unique identifier with which routing protocols and other applications can identify it. Routers are assumed to operate correctly and information is assumed to be stored without errors.

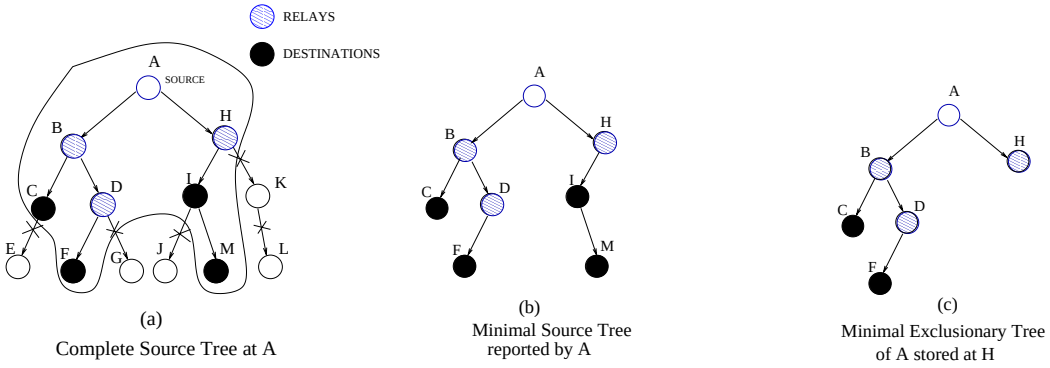


Figure 2.1: Routing information stored, communicated and processed in SOAR

2.1.1 Overview

Each router in SOAR maintains a source tree, which consists of the links that the router uses to reach known destinations. For example, as shown in Fig. 2.1(a), the source tree of router A contains links that it needs to reach destinations $B, C, D....L, M$. Let us assume that, among the known destinations, node A has active flows with nodes C, F, I, and M (black nodes in Fig. 2.1(a)).

Routers advertise to their neighbors the paths they take to reach *important* destinations. *Important* destinations are active destinations, relays to active destinations, or future potential relays of data packets. Using the example of Fig. 2.1, among the nodes in node A's source tree, nodes C, F, I, and M are active destinations; and B, D and H are relays required to deliver data packets to the active destinations. Therefore, router A does not report links (I, J), (H, K), (K, L), (D, G) and (C, E) (shown with cross-marks in Fig. 2.1(a)) that belongs to paths to destinations which are not important. It advertises all the other links, shown within the curved boundary in Fig. 2.1(a) that are in the paths for important destinations. The subset of the

source tree that a router advertises to its neighbors constitutes its *minimal* source tree (Fig. 2.1(b)).

Every node builds a partial topology of the network using the state of its outgoing links and the link-state information in the minimal source trees advertised by its neighbors. A path selection algorithm is then run on this partial topology for computing routes to known destinations.

2.1.2 Information Stored

Conceptually, each router i in SOAR maintains a partial topology table T_i, its source tree ST_i, a routing table RT_i (where each entry contains destination ID, the next hop for the destination) and the minimal source tree ST_i^x, reported by each neighbor x (ϵN_i) of node i , where N_i is the list of neighbors of node i. A router also keeps a query table, a data buffer and destination table (D_i).

For implementation purposes, a single data structure can incorporate all information present in ST_i^x, ST_i and T_i. For each link $l_u^v \in T_i$, information that needs to be stored are its head u, its tail v, its cost c, its list of neighbors who have reported l_u^v (which is essentially information about ST_i^x) and a flag denoting whether the links are part of node i's source tree (which is essentially information about ST_i).

Router i maintains the information about whether a destination j is *important* or not in a destination table D_i. An entry in D_i for destination j maintains a variable, $last\_heard_j$, which tells the last time when either one of the following events has happened:

- a data packet for node j is received at node i
- a reply containing a route to node j is received.
- a query is received at node i, which has been originated by node j.
- a new neighbor j is discovered.

When the difference between the present time and $last\_heard_j$ is greater than the SOAR-specific protocol parameter, *refreshing_time*, router i will no more consider node j as *important*, unless (a) node j is used as a relay for any other node k, and (b) node k is important.

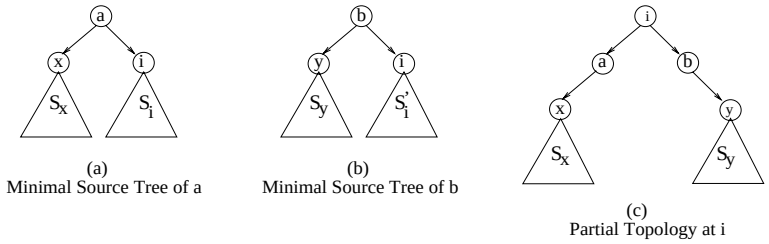


Figure 2.2: Building partial topology in SOAR using minimal source trees of neighbors

When a node i receives the source tree advertised by its neighbor k, it determines the subtree in node k's *minimal source tree* for which node i is the root and excludes the links belonging to that subtree and considers the remaining subset for inclusion in the topology database. This subset of the advertised minimal source tree that gets included in a router's topology is called the *minimal exclusionary* tree.

In the example of Fig. 2.1 , the minimal exclusionary tree of node A stored at node H corresponding to the minimal source tree (Fig. 2.1(b)) reported by node A to node H is as shown in Fig. 2.1(c). It should be noted that the *minimal exclusionary*

Table 2.1: Terminology for SOAR

l_u^v	:	Link entry in the topology for the link from node u to node v, it is a tuple $(u, v, cost, seqNo)$
l_{uk}^v	:	Link entry from node u to node v as advertised by neighbor k
L.cost		Cost of the link entry L
L.seqNo	:	Sequence number of link entry L
s_{uk}	:	highest sequence number advertised by node k to node i for destination u, same as l_{uk}^v.seqNo
G	:	**(V,E)** is the graph of the network according to an omniscient observer
D_i^j	:	Distance from node i to node j according to the view of node i
D_{ix}^j	:	Distance from node i to node j as known by the neighbor to node x.
ST_x^i	:	Source tree reported by node x to node i
P_{xi}^j	:	path of node x to node j as known to node i

tree of node A at node H does not include links that are in the subtree rooted at node H. The reason behind excluding these links is that node A is using node H to reach any node within the excluded subtree and node H will have more current information about the status of the links in that subtree.

The partial topology maintained at a router is created by aggregating the *minimal exclusionary* trees of all its neighbors. As shown in Fig. 2.2, node a has reported to node i subtrees S_x (rooted at node x) and S_i (rooted at node i) and links (a, x), (a, i), while node b has reported to node i subtrees S_y (rooted at node y) and $S_i^{'}$ (rooted at node i) and links (b, y), (b, i). Minimal exclusionary trees of nodes a and b and adjacent links of node i aggregate to form the partial topology at node i, as shown

in Fig. 2.1(c). (Subtrees rooted at node i, S_i and S_i' have been deleted from the final topology table stored at node i.) In case of conflicting information about the status of a link, advertised by different neighbors, sequence numbers are used to resolve the conflict. A node trusts the status of a link reported with a higher sequence number.

Sequence numbers in SOAR are managed on a per-node basis, rather than on a per-link basis and at any instant of time, the sequence number of any link is the sequence number of the head node of the link. Each router increments its sequence number when any of its outgoing links goes up or down, which implies the sequence number of a link with head node u can increase due to the change in link status of other adjacent links with the same head node u. Every time a node advertises its source tree, each of its outgoing links carry the current node sequence number.

The sequence numbers of links with the same head node u at a router i's database ($i \neq u$) can be different. This is because information about links with the same head node propagates towards a node from different directions and intermediate routers do not report all link-state changes to limit updates.

The following rules are used to add and delete links from the topology graph of a router (for terminology refer to Table. 2.1):

1. If a link l_u^v is deleted from a neighbor k's minimal source tree and (a) neighbor k advertises a link $l_u^x = (u, x, \infty, s_{uk})$, where $x \in V$ and (b) $s_{uk} > l_u^v.seqNo$, then the link-state entry l_u^v of the topology database is marked (u, v, ∞, s_{uk}). The reason behind this is the neighbor has advertised the highest sequence number of node u and has deleted l_u^v from its source tree.

2. If a neighbor k advertises a link $(u, v, cost, s_{uk})$, then $l_u^v.cost = cost$ and $l_u^v.seqNo = s_{uk}$, if $s_{uk} > l_{u'}^v.seqNo$ or $s_{uk} = l_{u'}^v.seqNo$ and $cost < l_{u'}^v.cost$ where $l_{u'}^v$ is the previous link-state entry in the topology.

Since sequence numbers to links are assigned from a finite number space, one important cause of concern would be how to achieve the roll-over of the sequence number. There are two simple ways in which roll-over of sequence numbers can be supported in SOAR to validate link-state updates. In one approach, an aging field is used in addition to the sequence number of a link-state update (LSU). The largest possible sequence number is sent with a zero age and each node is forced to delete the link from its tables and propagate such an LSU; furthermore after establishing a new link with a new neighbor, a node sends to its neighbor the last sequence number for the neighbor, so that the neighbor can start using a sequence number larger than such a value. Another approach consists of using a timestamp together with the sequence number. The timestamp is maintained externally to the algorithm, and eliminates the need for resetting the sequence number, because the timestamp increases monotonically.

2.1.3 Neighbor and Route Discovery

SOAR does not need to depend on a link-layer neighbor protocol for monitoring link connectivity with neighbors. When a router receives a control packet directly from a router that is not currently a neighbor, it assumes that a link with a new neighbor has been established. It is assumed that either a link-layer protocol can inform

SOAR about a link failure when data packets cannot be sent along that link, or SOAR can make that determination after a few transmissions to a neighbor.

When a router receives a data packet for which it is the next hop and the router has a valid path to the destination, it immediately forwards the packet. Otherwise, the router initiates a route-discovery process and keeps the data packet in its *data buffer* while the process completes.

The path-discovery cycle consists of two kinds of queries: non-propagating queries that are meant for neighbors only, and propagating queries that are flooded throughout the network.

Non-propagating queries are sent prior to the propagating queries to prevent unnecessary flooding when the neighbors of a router have a path to the required destination. Two path discovery cycles are separated by *query_send_timeout* seconds. The value of *query_send_timeout* is doubled each time a response is not obtained during a path-discovery cycle, until a pre-defined number of attempts have been made, after which it is kept constant. After a pre-defined number of attempts, the node can notify its upper layer that the destination is unreachable and it is up to the upper layer to decide whether to terminate the flow or not.

A query for a particular destination is forwarded by a node if all of the following conditions are satisfied:

- The router does not have a path to destination.
- The query has not traversed its stipulated number of hops.

- The difference between the present time and the time when the query for destination was last forwarded is greater than $query\_fwd\_timeout$.

The second condition is required to limit the queries within a specified zone. The last condition is imposed to limit the propagation of queries in the network when they are meant for the same destination and originated from different sources.

A node sends a *reply* to a query when it has a path to the destination requested, while it forwards a reply if all of the following conditions are satisfied:

- The router has a path to the destination of the reply.
- The router has discovered the new route to the source of the packet only after receiving the reply (this is required to prevent multiple replies).
- The router is a node in the path from the source to the destination (this prevents flooding of route replies).

The replies can be forwarded in two different ways: (1) they can be unicast to the originator of the route discovery process; or (2) they can be broadcast and only those neighbors who satisfy the third condition discussed above re-broadcast the route reply.

When a router receives a query and forwards it, it marks the source of the query as important, which implies the query packet will contain the links that constitute the path to the source. This is required to set up reverse paths towards source, which are used to forward replies back to the source. In addition, when a node sends or forwards a reply, it marks as important the destination for which route discovery has

been initiated, so that minimal source tree carries the path to the destination. After the path has been established the data packets are forwarded hop by hop. When the routes break, SOAR uses updates to set up alternate paths and notify neighbors of the changes in the forwarding paths. Apart from the *queries*, *replies* and *updates*, in SOAR the routers exchange two other control packets which are (a) forced updates and (b) forced replies. These packets are essential to force some relevant nodes in the network to set up paths for certain destinations. In Chapter 3.1, we will describe in details the design objective behind sending those packets.

2.1.4 Handling Link Failures

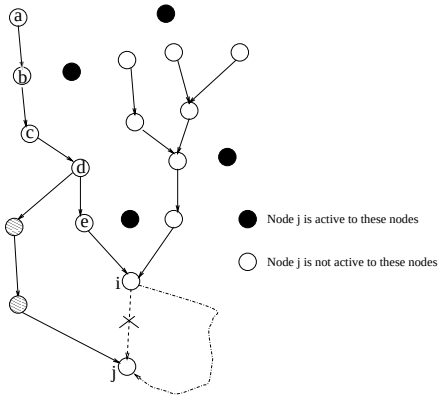


Figure 2.3: Propagation of link failure in SOAR

When a router detects that the cost of its path to the destination has increased, it sends updates to its neighbors. When costs of paths to *important* destinations decrease or remain the same, a node can change successor without notifying its neighbors, because this operation cannot lead to loops.

In Fig. 2.3, assume that active flows exist between node a and node j and

the path taken by the flow is $abcdeij$. Let us assume that link (i, j) fails. After node i detects the failure of link (i, j), it finds an alternate path (shown as the curved dashed line) to node i of higher cost and reports that path to its neighbors. The reported minimal source tree carries implicitly information about the deletion of link (i, j). The link failure information propagates to the upstream nodes, until it reaches a node (say d) that has an alternate equal cost path to node j. If the case would have been that none of the upstream nodes of node i has an alternate equal or lower cost path to node j, then the link failure information travels all the way back to the source, a. Under such scenarios, node a will initiate a route discovery process. Accordingly, intermediate nodes from source to receiver try to locally repair route failures, rather than always sending link failure information back to source. Unlike SOAR, in DSR and AODV route failure information travels always to the source. This implies that in SOAR the number of nodes who can get affected due to network dynamics is less compared to that in AODV or DSR.

The control packets exchanged in SOAR are primarily limited broadcast packets, which can be lost in a multi-access radio channel. Because the loss of control packets can lead to incorrect path information and loop formations, the path traced by a particular data packet is kept in its header. When a node receives a data packet to forward, it reads the path traversed by the packet in the packet header and checks whether forwarding it to the successor, specified in the routing table would lead to a loop. If it detects that the packet can go in a loop, it sends out an update and drops the packet. A router also sends an update if it receives a data packet for forwarding

for a destination to which it does not have a route.

To prevent an update from being sent for each data packet received from a burst of data packets with no next hop or headed for a potential loop, a *minimum update time* is enforced in the transmission of consecutive updates. This time spacing of updates is maintained only for those updates generated in response to information obtained through data packets.

2.1.5 SOAR Packet Format

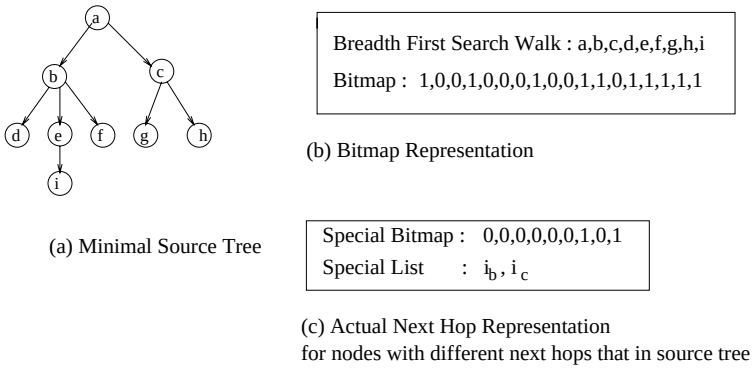


Figure 2.4: Representation of minimal source tree in control packets of SOAR

The minimal source tree can be efficiently represented using $O(4n + (2n - 1))$ bits by emulating the breadth-first search of the minimal source tree (n is the number of nodes in the network).

Fig. 2.4(b) shows the bitmap representation of the minimal source tree, given in Fig. 2.4(a). Breadth First Walk (BFSWALK) sequence shows the nodes in the order in which they are visited, if the search starts from the source. The bitmap in Fig. 2.4(b) will explain the parent-child relationship among nodes. After every "1", the following 0s are the children of that node, for whom the "1" has been placed.

Procedure *PACKET_DECRYPT* describes the process of re-creating a tree based on a given BFSWALK sequence and bitmap.

```
Procedure PACKET_DECRYPT
    count ← 0
    i ← 0
    while (count < bitmap.length)
        if (bitmap.bit[i] = 1)
            count ++
            parent ← BFSWALK[count-1]
            index ← count
            bitmap_index ← i + 1
            while (bitmap[bitmap_index] = 0)
                        u ← BFSWALK[index]
                        u.parent ← parent
            end                 i ← bitmap_index
        endif
    end
```

The number of bits in the bitmap is $(2 \times n - 1)$, where n is the number of nodes in the tree. [1]

Assume that according to the routing table entries, in order to reach node g, the path taken is acg, while for node i the path taken is $acbei$. Therefore, node i's and node g's next hops are different from what they have been shown in the minimal source tree in Fig. 2.4(a). The reason behind the difference in the actual next hop and that represented by the advertised source tree is that SOAR advertises source-tree only and the routes computed through the path selection algorithm combine to form a graph,

[1] Thanks to Marcelo Spohn for the efficient packet representation method.

rather than a tree. We will explore the issue in detail in the next chapter. In the SOAR control packets, actual next hops are advertised as shown in Fig. 2.4(c). The bits in the *special* bitmap correspond to the nodes shown in BFS Walk and marking the bit 1 in *special* bitmap implies that the corresponding node has different next hop than that indicated in the BFS walk. Their real next hops are indicated by the indices of those next hops in BFS walk (e.g., in Fig. 2.4(c) i_b is the index of node b in BFSWALK and node b is the actual next hop for node g). The total overhead (in bytes) for representing this special list is $(\lceil n/8 \rceil + \lceil log_2 n/8 \rceil \times m)$ where m is the number of nodes with different next hops than that shown in the minimal source tree.

2.1.6 Example of SOAR Operation

We illustrate the steps of SOAR operation using the example shown in Fig. 2.5. For simplicity, we assume that all the nodes have packets for every other node in the network and therefore, every other node in the network is *important* for each node and that the network has converged to the same sequence number for each node. Therefore, node a knows the highest sequence number for each node in the network. Corresponding to each node in the figure, its sequence number has been indicated in brackets, e.g. 34 is the current sequence number for node b and that is known to all nodes in the network. The example shows how the partial topology table at node a in Fig. 2.5 is modified after link (b, c) fails.

When link (b, c) goes down, node b increments its sequence number to 35. The path to node c breaks at node b and it sends an update reporting its new minimal

source tree, as shown in Fig. 2.5(f). Node a receives the update and modifies the minimal source tree entries of node b. No update has yet reached from node f and so the minimal exclusionary tree of node f at node a remains unchanged. The links deleted from the old minimal source tree of node b at node a are (b, c), (c, d), (c, e). Links (c, d) and (c, e) do not appear in the minimal source tree of node f. These links are deleted from the partial topology table at node a. The node sequence number reported by node b for itself is 35 and the node does not advertise link (b, c). Because every node must be using the shortest path to any destination, the only reason node b has stopped using the link (b, c) is that link (b, c) failed or increased in cost or node b does not use it because there is an alternate lower cost path. Link (b, c) advertised by node f has sequence number $34 < 35$. As indicated in Fig. 2.5(h), node a marks link (b, c) to be of infinite cost with sequence number 35. The reason for setting the cost to infinity is to stop using the link because the neighbor who has already stopped using it has advertised the highest sequence number for link (b, c). This technique helps to inform routers that a link may not be useful for data delivery without the explicit notification of link failures. The modified network topology, as shown in Fig. 2.5(h), is used for re-computation of routes.

2.1.7 Benefits of SOAR

We show using Fig. 2.6, how SOAR uses least number of control packets for setting up and maintaining paths by exchanging source tree information. Assume that initially the routing database of node i is empty. When node i wants to send a

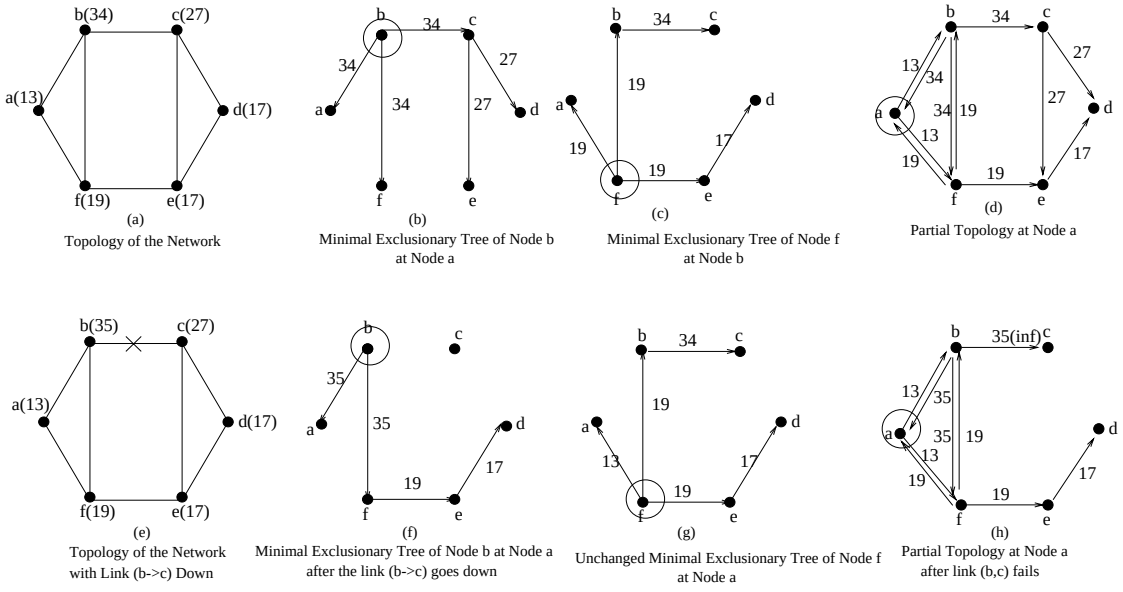


Figure 2.5: Handling link-state updates and updating network topology in SOAR

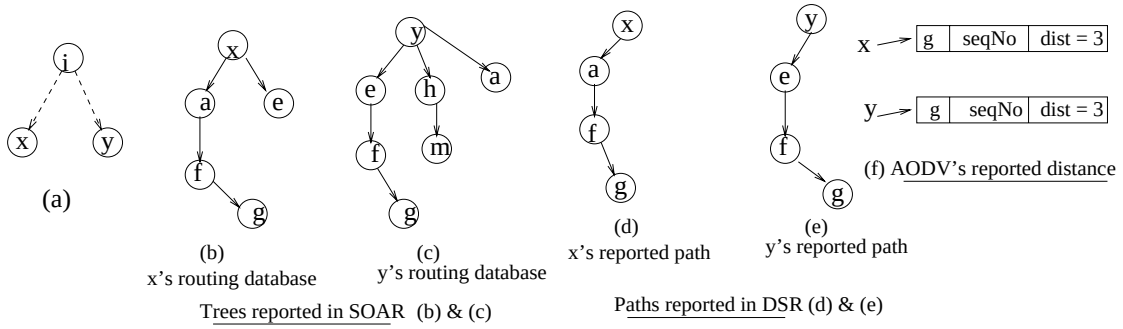


Figure 2.6: Difference in routing information exchanged in SOAR, DSR and AODV

packet to node g, it sends a query for node g because it does not have any path for it. Fig. 2.6(b) and Fig. 2.6(c) show the routing entries for active destinations at node x and node y, which are neighbors of node i. Nodes x and y reply to the query, because they both have routes for node g. In SOAR, nodes x and y will report minimal source trees, as shown in Fig. 2.6(b) and Fig. 2.6(c) respectively. DSR only advertises paths to the destination requested. The reported paths of nodes x and y in DSR are as shown in Fig. 2.6(d), while in AODV nodes x and y exchange distance information as

shown in Fig. 2.6(e).

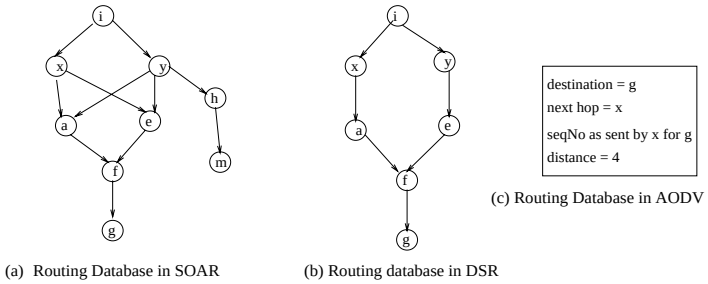


Figure 2.7: Building of routing database in SOAR, DSR and AODV on exchange of control packets

Fig. 2.7(a), Fig. 2.7(b), Fig. 2.7(c) respectively show the aggregated topology information in SOAR, DSR and AODV that gets stored at node i after the reception of replies to its queries. AODV has a single choice of route for node g because it accepts the first reply and discards the second. DSR accepts multiple replies and stores the information in each one of them and therefore, has two paths to node g, while SOAR, as shown in Fig. 2.7(a) has four choices to reach node g. Therefore in SOAR, when the original path breaks, the router can set up an alternate path more often than other routing protocols, without starting a fresh route discovery process.

In [21], a new version of DSR has been proposed in which a link-state database is created at every node by extracting links from each path information it receives and a path-selection algorithm is run on this database for final route selection. In SOAR links are updated, maintained and exchanged, while they are derived from path updates in DSR. SOAR communicates a link *only once* when it is used to reach at least one destination and validates each link with a sequence number. In contrast, DSR with link caches [21] specifies complete paths to destinations from which links are

then extracted; links are not validated individually and are deleted by aging. Though the link-state representation of paths in DSR helps it to improve the performance, the source-tree based method of exchange in SOAR always helps it to create a richer routing database than DSR.

2.2 Correctness of SOAR

The correctness of an on-demand routing protocol must be approached differently from the correctness proofs of table-driven protocols in which each node maintains routing information for all destinations. The key reasons behind the need for a different approach are that different routers maintain routing information for different subsets of destinations, and routing information at a router is modified due to changes in network flows in addition to changes in the network connectivity.

For a table-driven (proactive) routing protocol to be correct, it must satisfy the following two properties:

- All nodes have correct paths for any given destination j following an arbitrary number of changes in the network conditions.
- No update messages are being exchanged a finite time after the last change in the network condition.

What need to be shown for the correctness of on-demand routing protocols are:

- Within a finite time following an arbitrary number of changes in network condi-

tions and traffic flows, all sources have correct paths for each destination that is reachable through a physical path and for which they have an active flow.

- Within a finite time after the last change in network conditions and flows, all sources have loop-free and valid paths to all reachable important destinations, and no finite paths to important unreachable destinations.

The above conditions leave open the possibility of nodes only trying to find paths to unreachable destinations. In practice, the routing protocol can inform that a destination appears to be unreachable after a few failed attempts, and it is up to the higher-level protocol or application to determine whether or not to persist trying to find paths to unreachable destinations.

The assumptions made to prove the correctness of SOAR are:

1. After an arbitrary sequence of link cost changes, topology changes, and flow changes, there is a finite amount of time that is sufficiently long for SOAR to finish the final computations of routes to important destinations.

2. Control messages are received correctly within a finite time, or nodes have the ability to detect loops from information carried in data packets.

3. Routing information is stored correctly and routers operate correctly.

4. An underlying protocol monitors neighbor connectivity and within a finite time after the last network-connectivity change, every node knows which are its neighboring nodes.

Theorem 1 *Within a finite time after the last change in the topology or network flows, no updates are transmitted.*

Proof : If the updates are transmitted indefinitely after the last change in network connectivity or network flow, then there must be at least one node, which is sending updates indefinitely for at least one link. We will show by induction that this is not possible for any link in the network.

Let d_l be the number of hops that a node is away from the head node of a link l.

If $d_l = 0$, it means we are referring to the head node. The head node changes the status of a link only finite number of times because there are finite changes in network connectivity. (In SOAR, a node changes sequence number only if there is a change in the status of any of its adjacent links). Therefore, the proof holds true for the head node.

The induction hypothesis is that a node which is n hops away from head node of a link will not send updates indefinitely for that link.

Let us now consider the case of a node that is $(n + 1)$ hops away from the head node of the link l. If a node sends updates indefinitely for a link l, then it must add or delete the link indefinitely from its source tree, in which case one of its neighbors has added or deleted the link indefinitely from its source tree. Moreover, that neighbor is one hop nearer to the head of link l (i.e. n hops away) because due to the use of exclusionary trees, a node does not trust links advertised by the upstream nodes, i.e., nodes that are farther away. But according to induction hypothesis, the

downstream node, which is n hops away from the head node will not add or delete the link indefinitely. Therefore, the node, which is $(n+1)$ hops away will not add or delete the link indefinitely and the updates cannot go on for infinite time.

The above proof does not make any assumption regarding the reliability of any particular update and hence the proof should be valid under unreliable updates.

Theorem 2 *SOAR does not have permanent loops.*

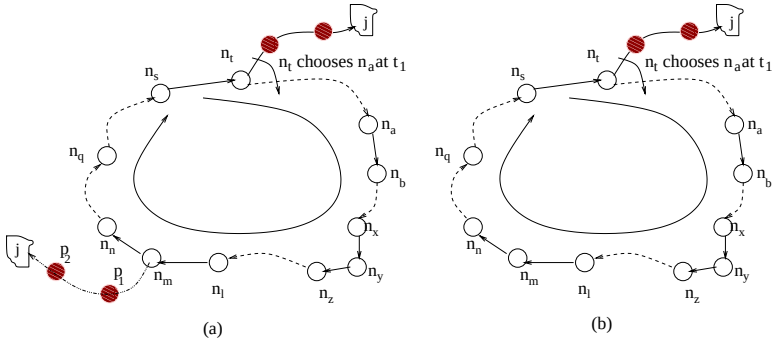


Figure 2.8: A temporary loop in SOAR

Proof : We will divide the proof into several stages.

Assume initially that every destination is important to every other node in the network, i.e., traffic for any destination is present at any node all the time. This implies that the source tree reported by any node contains links to all other nodes in the network.

As shown in Fig. 2.8(a), let us assume that nodes n_a, n_b, ...,n_x, n_y, n_z, ...,n_l, n_m, n_n, ..., n_q,..., n_s, n_t form a loop, and that the loop is formed at time t_1 when node n_t chooses node n_a to reach some destination j. The loop is formed because node n_a is actually an upstream neighbor for node n_t, though that information is not

available at node n_t. This is because the source tree advertised last by node n_a to node n_t does not include the subpath from node n_t to destination j. That can happen if node n_a does not know of its latest path to node j or chooses not to advertise its latest path.

Let node n_m be the node nearest to node n_a that has not advertised its latest change in successor in its path to node j (node n_m can also be equal to node n_a, but $n_m \neq n_t$, because then node n_t should have known about the correct path to node j through node n_a).

When node n_m changed from its previous successor (p_1) to node n_n, it must have experienced a decrease in cost. This is the case because, if node n_m had experienced an increase in cost, it would have advertised its latest change in successor, in which case the assumption that node n_m is the nearest node to node n_a who has remained silent after changing successor will not hold.

By recursion, while proceeding clockwise along the loop, either a node n_q is reached that has not advertised its latest change in successor (due to decrease or no change in path cost for node j) or node n_t is reached. The last condition cannot happen, as that contradicts our assumption.

Let us now assume that n_m is the node extreme from n_a that has remained silent due to decrease or no change in its path cost. This implies that n_m is the last node that knows of the correct path through node n_t to destination j. Accordingly, the following equations should be true for node n_m and its downstream nodes for node j (for terminology refer to Table 2.1):

$$D^j_{n_m}(t_0) > D^j_{n_n}(t_0) > ..... > D^j_{n_t}(t_0) \qquad t_0 < t_1 \tag{2.1}$$

If node n_m's path cost to node j decreased or remained the same, then for any node x upstream to node n_m , $D^j_{n_m x} \geq D^j_{n_m}$. Therefore, we have

$$D^j_{n_m n_l}(t_0) \geq D^j_{n_m}(t_0) \Longrightarrow D^j_{n_l}(t_0) > D^j_{n_t}(t_0) \tag{2.2}$$

Extending this, it is easy to show that

$$D^j_{n_a}(t_0) \geq D^j_{n_l}(t_0) > D^j_{n_t}(t_0) \tag{2.3}$$

$$\Rightarrow D^j_{n_a n_t}(t_0) \geq D^j_{n_a}(t_0) > D^j_{n_t}(t_0) \tag{2.4}$$

$$D^j_{n_t}(t_1) \geq D^j_{n_a n_t}(t_1) \geq D^j_{n_q}(t_0) \geq D^j_{n_m}(t_0) > D^j_{n_t}(t_0) \tag{2.5}$$

Eq. 2.5 implies that node n_t must have an increase in cost when it chooses n_a as the successor at t_1. The above steps demonstrate initially a temporary loop can only form if a node chooses a successor which is presently an upstream node, if it experiences an increase in cost. Then it advertises path $[l^{n_a}_{n_t}, l^{n_a}_{n_b}....P_{n_y n_x}]$ and the cost of the path is greater than $D^j_{n_a}(t_0)$. From Eq. 2.5, this implies that all nodes $n_s, ..., n_q, ..., n_m, ...n_z$ must experience an increase in cost and send updates, provided none of them has found an alternate path of equal cost without involving subpath from node n_t, in which case the loop breaks. When node n_y receives the update from node n_z advertising higher cost, it will not choose n_z as the next hop as node n_z's advertised path ($[l^{n_t}_{n_s}....l^{n_m}_{n_l}...l^{n_a}_{n_t}, l^{n_a}_{n_b}....P_{n_y n_x}]$) would indicate that the route to node j contains node n_y and the loop would break. That leads us to the first part of the

proof, where we show the permanent loops cannot happen if every node is important for every other node. The proof assumes that the control packet sent by a node will be received in finite time by the intended recipient.

When a node stops assuming a destination j to be important, updates are not sent for the change of paths it may encounter for destination j. We have seen that permanent loops can never be formed if the nodes change path due to decrease in cost. Next, we prove that permanent loops cannot occur if the distance increases and path changes have not been reported because the destination is not *important*. For this case, we need to consider two cases: reliable transmission of updates and unreliable transmission of updates. Under the assumption of reliable updates, we shall show that two hop loop detection mechanism should be sufficient to prevent permanent loops. For unreliable updates, a mechanism should be present to detect multi-hop loops.

When a node i detects the formation of a two-hop loop, it sends an update and in the update it specifies the path through the neighbor k. Then node k finds that its downstream neighbor i is actually using it to reach node j and hence will not select node i as the next hop and this will break the loop. Therefore, permanent two-hop loops cannot form.

For this case, we use the example, shown in Fig. 2.8(b). At time t_1, node n_t chooses node n_a as its next hop. Let the distance from node n_t to node j be c_0 when node n_t has a different next hop than node n_a. Let n_l be the first node in the counter clockwise direction that has not advertised its increase of cost and let n_x be the node, till which node n_a knows correctly the data path. Node n_m is the downstream

neighbor who does not know that node n_l has chosen it as the successor for node j. Node n_m's current successor is node n_n and not node n_l, which leads to Eq. 2.6

$$D^j_{n_l *} = D^j_{n_l n_m}(t_0) \geq D^j_{n_n n_m}(t_0) \geq (c_0 + \alpha - 1) \textbf{ where } * \in N_{n_l} \tag{2.6}$$

The assumption is that $D^j_{n_m}(t_0) = c_0 + \alpha$ $(t_0 < t_1)$ and the cost metric of a path is the hop count $(\alpha > 0)$.

Next we show that although under the assumption some node n_l has remained silent, the upstream nodes for destination j will have a lower estimate of their distance to node j and the estimate also has a lower bound.

Following Eq. 2.6 if node x is an upstream node β hops away from node n_l, then the following equation is true

$$D^j_{n_l}(t_0) \geq (c_0 + \alpha - 1) \Longrightarrow D^j_x(t_0) \geq (c_0 + \alpha + \beta - 1) \tag{2.7}$$

Using the same argument, it is true that $D^j_{n_a n_t}(t_0) \geq (c_0 + \alpha + \beta - 1)$. When $\alpha + \beta = 0$ and the equality holds, it implies that the loop is a two-hop loop, and it has already been shown two-hop loops do not exist permanently.

When $(\alpha + \beta)$ is a finite value, we have the following relation:

$$D^j_{n_t}(t_1) = D^j_{n_a n_t}(t_0) + 1 = c_0 + \alpha + \beta > D^j_{n_t}(t_0) \tag{2.8}$$

Accordingly, when node n_t chooses node n_a at time t_1, it experiences an increase in cost and it must send an update and the loop breaks, because the downstream nodes also experience an increase in path cost.

In case of unreliable updates, if node n_t's updates are lost, node n_t looses the opportunity to update its neighbors when it chooses node n_a as the next hop.

If a multi-hop loop detection mechanism is used, then node n_t can again update its neighbors.

Lemma 1 *If a source has a path to a destination and according to the network topology, **G** as seen by an omniscient observer that path is invalid, then data packets stop flowing along that path within a finite time.*

Proof : Assume that the source i has a path to the destination j and according to **G** that path is invalid. Because node i is the source for node j, within a finite time after node i has chosen that route, it must have data packets forwarded towards node j along that route. Given that the number of nodes in the network is finite, if the path is invalid, then the packet should either revisit a node (implying a loop) indefinitely, or visit a node that has no path to the destination indefinitely. From the previous theorem, the first case cannot occur. The rest of the proof shows that the second condition is also impossible.

Let i, n_a, n_b, n_c ..., n_x, n_y be the sequence of nodes along which the data packets flow and node n_y is the node which has no finite path for the destination j. In SOAR, if a node has a path for node j through neighbor k, it implies that node k has advertised that path to destination j. Because node n_x forwards data packets for node j to node n_y, $ST_{n_y}^{n_x}$ must contain links that lead to node j, which implies that node n_y has not advertised that it has no path to node j. However, this contradicts the operation of SOAR, in which any node that receives a data packet to forward and has no path to node j must send an update and that update would not contain any path to node.

The previous Lemma is also valid if the updates are unreliable, because if $l_{n_y}^{n_x}$ exists, then there is a finite probability that the update of n_y reaches n_x.

Lemma 2 *If there exists a path from source i to destination j according to $\boldsymbol{G}$, then node i can obtain a path to node j within a finite time.*

Proof : When a source i has no path to the destination j, it generates queries to set paths to destinations. If a path from node i to node j exists in **G** and node i does not get a path to node j within finite time, it implies that node i generates infinite queries to the destination, which is possible only if:

- No node has sent a reply, which cannot be true because at least node j should send a reply.
- At least one node sends indefinitely wrong replies. If a reply is wrong either
    1. The source i receives an invalid path. However, from Lemma 1, it follows that the node sending the wrong reply will be corrected.
    2. The source or the intermediate nodes that have forwarded queries and received wrong replies can not obtain any path for destination j. According to the operation of SOAR, those nodes will correct the sender of the wrong reply. When updates are unreliable, assuming that there is a finite probability of packets going through, the sender of a wrong reply will get corrected within a finite time. Therefore, this process cannot continue indefinitely.

Theorem 3 *Within a finite time after the last change in topology and traffic flows, all sources will have loop-free and valid paths to all receivers.*

Proof : This follows automatically from the combination of Lemma 1, Theorem 2 and Lemma 2.

Theorem 4 *If any destination becomes unreachable, no source can have a finite path to any of those destinations within a finite time.*

Proof : The result is immediate from the previous theorem. When a node does not find any path to a destination it will indefinitely send queries. In such a case, the network layer can send an indication to the application layer after several attempts of route discovery and it is up to the application to terminate the connection and resume it later.

2.3 Comparative Analysis

The performance evaluation has been done in the ns2 simulation platform [18], using the code of DSR and AODV provided with the simulator. For AODV, we have used the code available from Marina et. al. [27]. The AODV code conforms to the specifications mentioned in the version 3 of the Internet draft of AODV. However, the values of constants used for AODV are according to the values given in the code. DSR code conforms to the version 1 of the Internet draft. The link layer implements the IEEE802.11 [5] Distributed Co-ordination function (DCF) for wireless LANs. The broadcast packets are sent unreliably and are prone to collisions. The physical layer approximates the 2 Mbps DSSS radio interface of a Lucent WaveLan Direct-Sequence Spread-Spectrum radio [46]. The radio range is 250m and for all the simulations, the

run length is 900 seconds.

DSR, AODV and SOAR use link layer indications about the failure of links when data packets cannot be delivered along a particular link. Except for the notification of the link layer about links going down, none of the protocols has any other interaction with the lower layer. In particular, promiscuous listening was disabled for both DSR and SOAR. Given that both SOAR and DSR carry path information in data packets, promiscuous listening could improve the performance of these two protocols. However, our performance comparison of SOAR with DSR and AODV is aimed at evaluating the three protocols without making too many assumptions regarding the level of control of the MAC layer. Furthermore, in practice, promiscuous listening may not be available because of any of the following reasons:

- Link-level encryption may be used, which eliminates promiscuous listening.
- The MAC protocol uses multiple channels rather than a single common channel.
- Promiscuous listening is turned off to reduce energy consumption [11].

Mobility Pattern The movement of the nodes in the simulation is according to the random waypoint model [9], which we use to offer a common reference point with prior published results [8, 9]. In this model, each node is at a random point at the start of the simulation and after *pause time* seconds selects a random destination and moves to that destination at a speed selected from a uniform probability distribution between 0m/s and 20m/s. distributed between 0m/s and 20m/s. Upon reaching the destination, the node pauses again for *pause time* seconds, chooses another destination, and proceeds

there. For our simulation studies, we use moderate sized networks consisting of 20 nodes moving over a rectangular space of 1000m×300m.

Pause times used are 0, 15, 30, 45, 60, 120, 300, 600, and 900 seconds. We focus on high mobility scenarios with higher granularity than on low mobility scenarios, because our aim is to find how the routing protocols impart extra overhead under rapidly changing network conditions.

Input Traffic Pattern For most of the scenarios we have used the network traffic load of 40 packets/s or approximately 163 kbps with a data packet size of 512 bytes), which is small compared to the available bandwidth. The motivation behind using such a small load is to avoid having data packets compete with the routing layer packets, and to find out whether the control packets themselves create any congestion in the network.

In ns2, when the links go down all packets scheduled over that link can be rerouted. Therefore, when the load is less, we are testing whether the routing protocols can reroute data packets in case of link failures, or whether they loose packets due to formation of loops and overflow of data buffers at the routing layer where the data packets wait for their routes to be discovered using the route requests.

We have also tested the effect of overloading the network on the performance of routing protocols by varying packet loads from 20 packets/s to 140 packets/s.

Under increasing number of peer-to-peer connections, the challenge of on-demand routing protocols is maintenance of increasing routing information. Therefore,

we have also determined how each protocol scales with the number of flows. Accordingly, we carried out experiments for 4, 10, 20 and 32 flows. The packet generation rate for each individual flow is changed to keep the total load constant when the number of traffic flows in the network increases.

Each flow is a peer-to-peer constant bit rate (CBR) flow and the data packet size is kept constant at 512 bytes. Each flow continues for 200 seconds and after the termination of the flow, within 1 second, the source randomly chooses another destination and starts another flow, which again lasts for 200 seconds.

Comparison Criterion The following performance metrics are used to compare the performance of the routing protocols:

- *Packet delivery ratio*: The ratio between the number of packets sent out by the sender application and the number of packets correctly received by the corresponding peer application.

- *Control packet overhead*: The total number of control packets sent out during the simulation. Each limited broadcast packet is counted as a single packet. Low control packet overhead is desirable in low-bandwidth wireless environments.

- *Average hop count*: The average number of hops the data packet took from the sender to the receiver. Smaller hop count implies that the routing protocol is using shorter paths to the destinations, thereby more efficiently utilizing the network resources.

Table 2.2: Constants used in SOAR simulation

query send timeout (exponentially backed off) (ms)	500
Zero query send timeout (ms)	30
Max number of pending packets	50
query forward timeout (s)	1
max_query_send_timeout (s)	10
Minimum Update Time (s)	3
MAX_HOPS	17
refreshing_time (s)	60

- *Average end-to-end delay*: The end-to-end delay measures the delay a packet suffers between leaving the sender application and arriving at the receiver application. This includes delays caused by route discovery latency, delay due to waiting at IP and MAC layers and propagation delays.

- *Byte overhead in routing packets*: The total number of bits used in routing packets. We have mentioned the results obtained for byte overhead. However, no graph has been attached in the following section.

Simulation Constants The constants for the different parameters of SOAR, DSR and AODV are shown in Tables 2.2, 2.3 and 2.4. The values for constants used in the public releases of AODV [27] and DSR [18] have been used for simulation. For SOAR, the constants common with other routing protocols have been chosen to be the same.

2.3.1 Effect of Promiscuous Listening

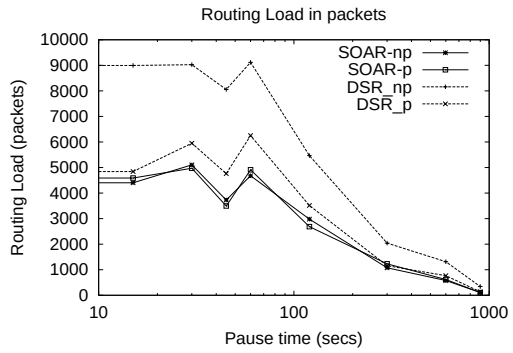
Promiscuous listening or snooping helps routing protocols to gather extra information without actively introducing any control overhead. When an interface is

Table 2.3: Constants used in DSR simulation

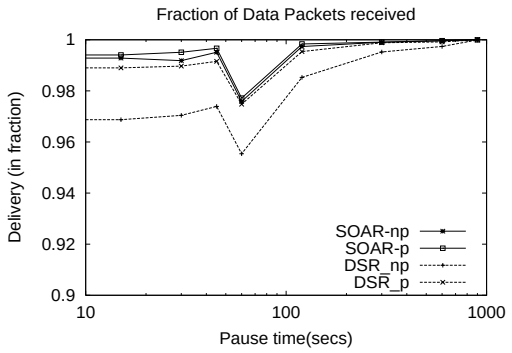
Time between Route Requests (exponentially backed off) (ms)	500
Size of source route header carrying carrying n addresses (bytes)	$4n+4$
Timeout for Ring 0 search (ms)	30
Time to hold packets awaiting routes (s)	30
Max number of pending packets	50
rt_rq_max_period(s)	10
Time to hold packets awaiting routes (s)	30
grat_hold_down_time(s)	1.0
max_err_hold(s)	1.0

Table 2.4: Constants used in AODV simulation

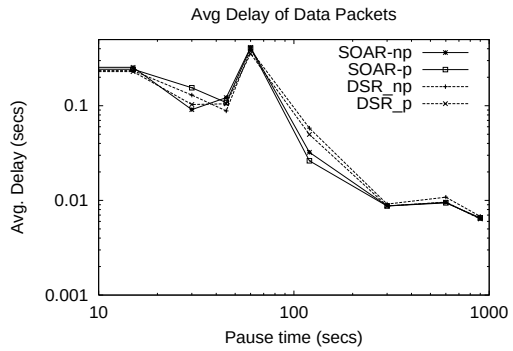
bcast_id_save (s)	6
max_rreq_timeout (s)	10.0
ttl_start	1
ttl_threshold	7
ttl_increment	2
node_traversal_time (s)	0.03
local_repair_wait_time (s)	0.15
network_diameter	30
rrep_wait_time (s)	(3 x node_traversal_time x network_diameter)



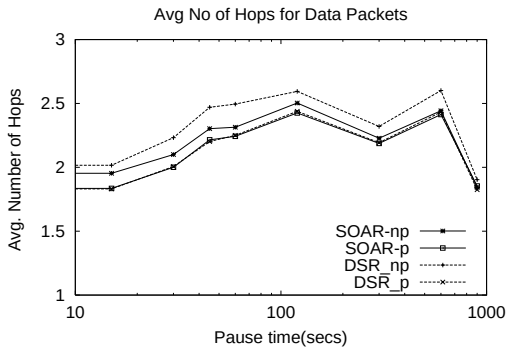
(a) Number of control packets produced



(b) Fraction of data packets received



(c) Average delay of data packets



(d) Average number of hops traversed

Figure 2.9: Effect of promiscuous listening (snooping) in a 20-node network with 20 flows for varying pause time

in promiscuous receive mode, every received packet is delivered to the network layer without doing any packet filtering.

We tested the effect of snooping in SOAR and DSR for a 20-node network with 20 flows under varying *pause time* and data load of 40 packets/s. In Fig. 2.9, *p* refers to the protocol with promiscuous listening, while *np* refers to the protocol without it. As Fig. 2.9(a) shows, DSR, rather than SOAR benefits more from snooping in terms of control packet overhead. This is because most of DSR's control packets are replies, which are unicast routing packets and so promiscuous listening enables the routers to gather more information. In contrast, most control packets exchanged in SOAR are broadcast packets, and hence snooping does not add much to the routing information gathered through ordinary exchange. Though DSR's control packets are significantly reduced due to snooping, *DSR-p* produces more control packets than either *SOAR-p* or *SOAR-np*, which shows SOAR's method of information exchange always gives an advantage over DSR.

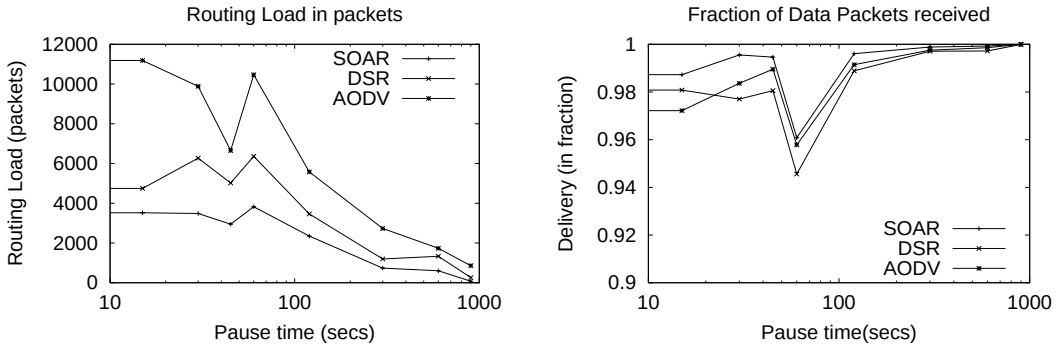
We observe from Fig. 2.9(d) that in terms of average path lengths both DSR and SOAR benefit significantly. DSR or SOAR does not depend on any neighbor protocol to detect the presence of new neighbors and hence promiscuous listening helps to discover neighbors faster and achieve route shortening. (*DSR-p* and *SOAR-p* has overlapping plot for average path length). This is also a reason behind improvement in data delivery due to snooping because the paths traversed become shorter and more up to date; therefore, the probability of packets getting dropped due to stale routing information reduces (Fig. 2.9b). The *e-t-e* delay (Fig. 2.9c) remains unchanged.

Promiscuous listening improves the performance of both DSR and SOAR, though with different degrees. However implementing promiscuous listening in practice faces several obstacles: Security considerations may prohibit the use of promiscuous listening and using encryption makes it very difficult to enforce it [10], promiscuous listening cannot be used accurately in networks with more than one common channel, and the energy consumption for promiscuous mode of operation is much higher than the energy consumption while discarding packets at the link layer [11].

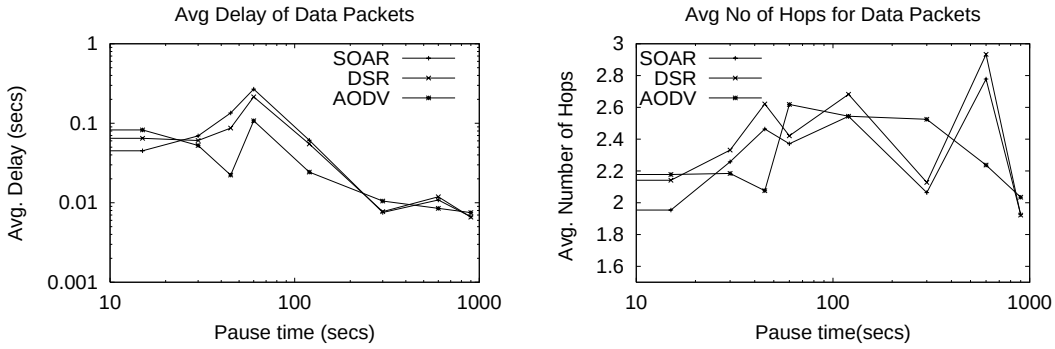
Given that DSR and SOAR both benefit from promiscuous listening, and because of the issues regarding its implementation and the fact that AODV does not rely on promiscuous listening, the rest of our comparison does not use snooping in DSR and SOAR.

2.3.2 Effect of Node Mobility

Fig. 2.10 and Fig. 2.11 present the effect of mobility of nodes on the performance of AODV, DSR and SOAR using a 20-node network with 10 and 20 flows and varying pause times and a traffic load of 40 packets/s. Fig. 2.10 and Fig. 2.11 show a discontinuity at the pause time of 60 seconds. This is because in that scenario the network gets partitioned and fewer data packets are delivered. Also the control packets are fewer in number because they consist mainly of queries, which are sent at long time intervals to re-discover routes. The graphs will be smooth if we have removed this scenario. However, we have not done so because network partitioning is a common scenario in ad-hoc networks and this scenario affects each protocol equally.



(a) Number of control packets produced (b) Fraction of data packets received

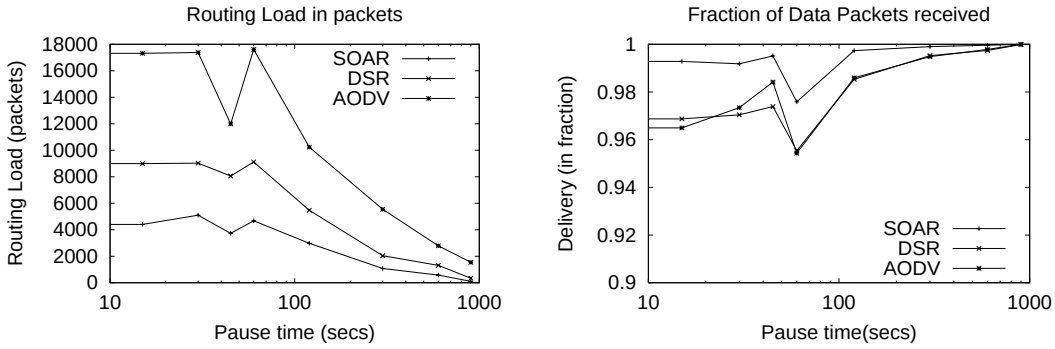


(c) Average delay of data packets (d) Average number of hops traversed

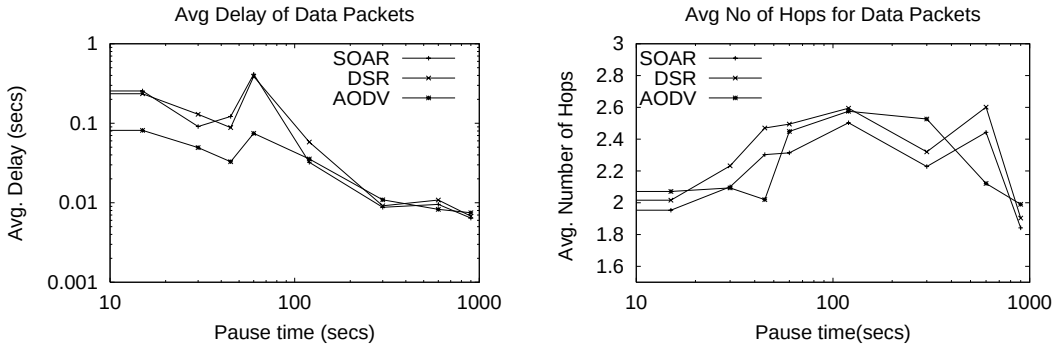
Figure 2.10: Effect of node mobility in a 20-node network with 10 flows for varying pause time

From Fig. 2.10(b) and Fig. 2.11(b) we see that, in all scenarios, all the routing protocols deliver above 90% of data packets at high mobility, while at lower mobility they deliver about 100% of data packets. However, we observe that SOAR has a better delivery rate than DSR and AODV under high mobility scenarios, while at lower mobility the delivery rate is almost the same for all protocols. In high-mobility scenarios, when a link breaks SOAR has more redundant paths to choose from, which means that it needs to drop fewer packets than DSR and AODV for non-availability of routes at forwarding nodes.

From Fig. 2.10(a) and Fig. 2.11(a) we see that the routing load in terms



(a) Number of control packets produced (b) Fraction of data packets received



(c) Average delay of data packets (d) Average number of hops traversed

Figure 2.11: Effect of node mobility in a 20-node network with 20 flows for varying pause time

of control packets is the smallest in SOAR, while AODV in all cases produces the largest load. Because AODV maintains a single path to a destination, it starts a route discovery process more often than SOAR or DSR. The difference in routing overhead among the three protocols is most pronounced for high mobility scenarios, and the difference reduces when the nodes are less mobile. SOAR during link breakage can have more alternate paths than DSR or AODV and hence resorts to less route discoveries. However, in terms of byte overhead DSR and SOAR give comparable performance. Although in AODV the size of control packets is smaller than in DSR and SOAR, the number of packets actually sent is too large to give it an advantage in

byte overhead.

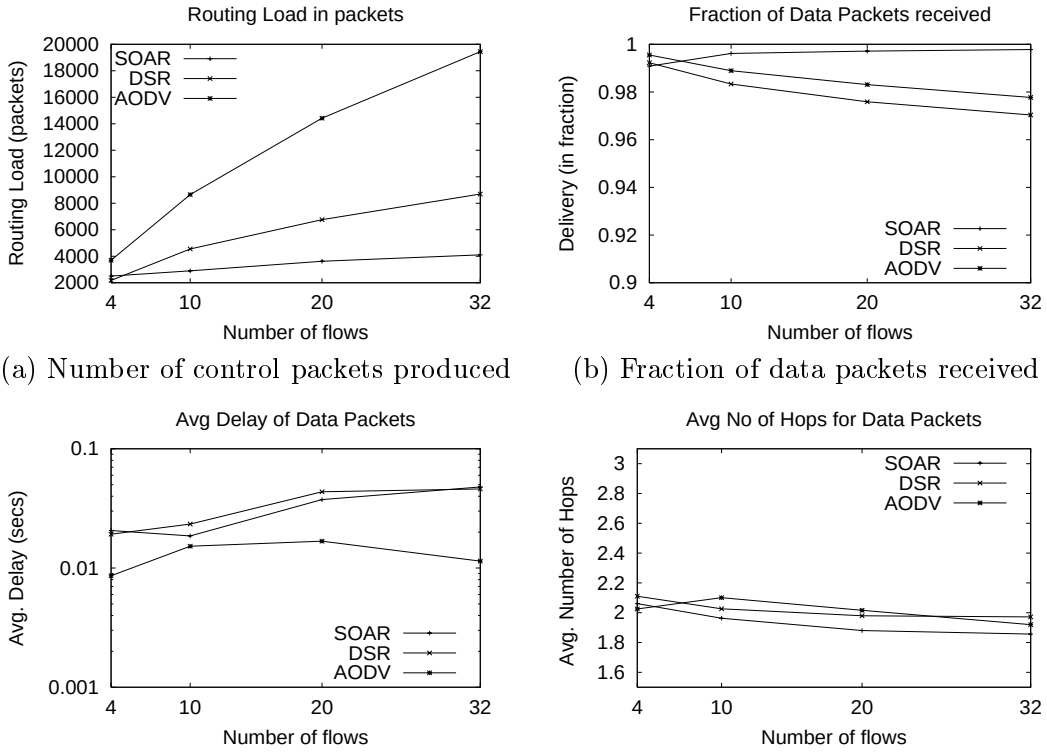
As can be seen from Fig. 2.10(d) and Fig. 2.11(d) SOAR has smaller average hop count than DSR on all situations, while AODV's average hop count results are mixed. SOAR and DSR both use the hop count as a metric for choosing routes, while the route discovery mechanism of AODV leads to the choice of the least congested path [8]. Therefore, the trend of the graph for average hop count for AODV is different from SOAR and DSR. SOAR usually has a shorter hop count than DSR. Exchange of source trees in SOAR helps to shorten routes without any promiscuous listening. Intuitively, the difference would be smaller between SOAR and DSR with promiscuous listening enabled in both.

We observe from Fig. 2.10(c) and Fig. 2.11(c) that, in most cases, AODV has better delay performance than SOAR and DSR. One reason behind this is that AODV accepts the route reply received first, while DSR or SOAR uses hop counts to choose among multiple options. Even though SOAR delivers more packets than AODV, it incurs longer delays because data packets explore alternate paths, thereby spending more time *queueing*.

2.3.3 Effect of Increase of Flows

On-demand routing protocols maintain paths to destinations as needed, so with the increase of the number of flows, more routes are required to be set up and hence the routing overhead increases.

Fig. 2.12, Fig. 2.13 show the impact of the number of flows on protocol



(a) Number of control packets produced

(b) Fraction of data packets received

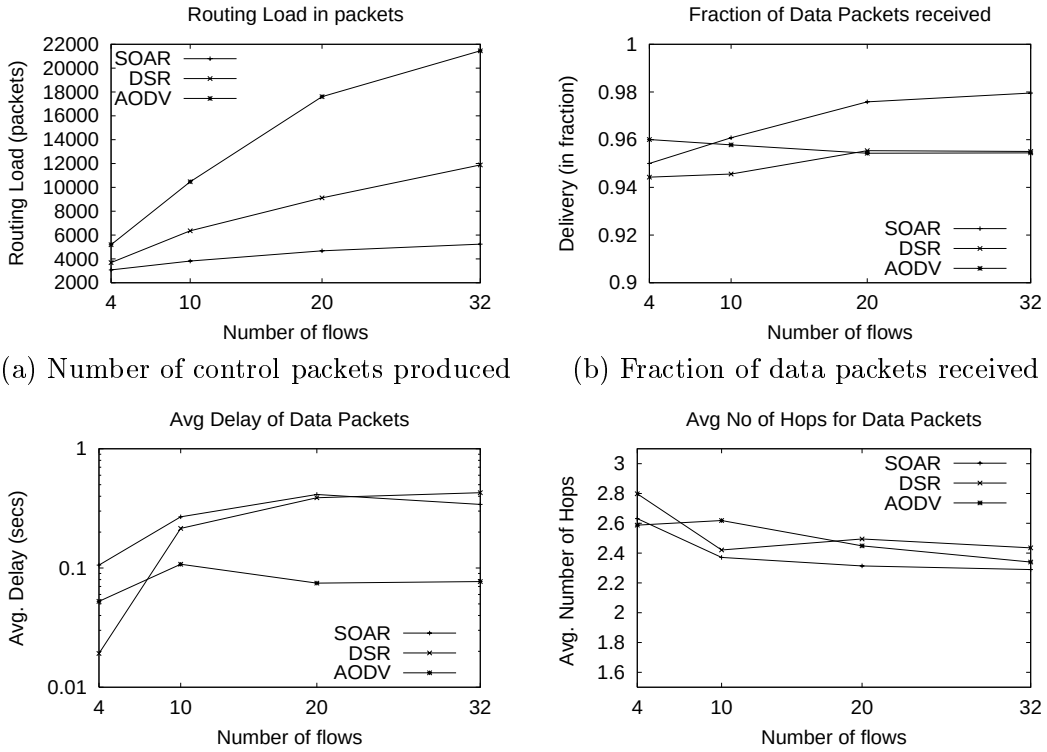
(c) Average delay of data packets

(d) Average number of hops traversed

Figure 2.12: Effect of increasing number of flows in a 20-node network for pause time 0 s

performance. The results shown are for a 20-node network with 4, 10, 20 and 32 flows when the pause times are 0 seconds and 60 s, and the packet load is 40 packets/s.

As we see from Fig. 2.12(a) and Fig. 2.13(a), the control packet overhead increases for each protocol with the increase of flows though the rate of increase is different in each one of them. In the high mobility scenario (*pause* time = 0 s, number of flows = 4), as shown in Fig. 2.12(a), DSR has slightly lower control overhead than SOAR. This is because SOAR has higher redundancy in its routing database. Maintaining redundant information helps SOAR, when the number of flows increases. As the number of flows increases, the number of control packets in SOAR increases



(a) Number of control packets produced

(b) Fraction of data packets received

(c) Average delay of data packets

(d) Average number of hops traversed

Figure 2.13: Effect of increasing number of flows in a 20-node network for pause time 60 s

at a much slower rate than both DSR and AODV. When the number of flows is 32, SOAR produces almost ten times fewer control packets than AODV and 2.7 times fewer than DSR.

The byte overhead in the routing packets of both SOAR and DSR are similar for all flows. This is because SOAR transfers fewer control packets with an increasing number of flows, but the size of its minimal source trees increases. The difference in byte overhead in control packets between AODV and the other two protocols become more pronounced when the number of flows increases, with AODV producing higher byte overhead all the time.

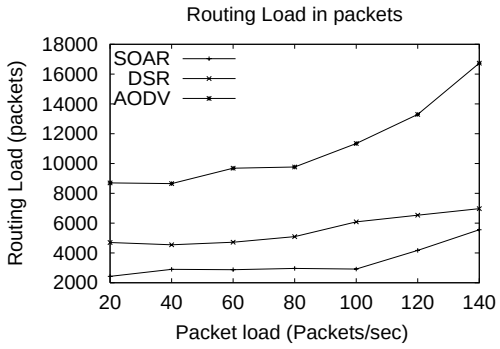
In terms of data delivery (Fig. 2.12(b) and Fig. 2.13(b)) SOAR delivers more packets than DSR or AODV when the number of flows is higher than four, while AODV delivers more packets when the number of flows is four. When the number of flows is less, less number of links are used for data delivery. Link failures are detected only during data transfer, hence when the number of flows is less, the path information can become more stale. As AODV does frequent route discovery, it finds more recent paths. But with the increase of number of flows, as more links are involved in data forwarding, routes in SOAR stay more up-to-date and hence they help to achieve higher data delivery.

We observe from Fig. 2.12(d) and Fig. 2.13(d) that SOAR provides shorter path length in most of the cases. The reason behind this is, as explained before, SOAR can advertise shorter routes for certain destinations when it decides to advertise longer routes for other destinations.

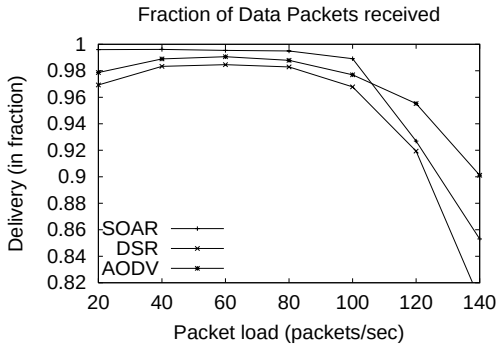
From Fig. 2.12(c) and Fig. 2.13(c), we see AODV has a shorter average delay than SOAR or DSR, because of less queueing at the link layer.

2.3.4 Effect of Network Load

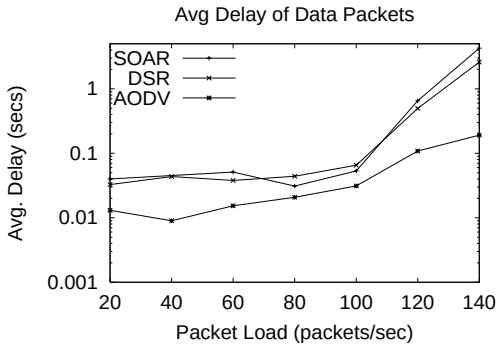
When the network gets more loaded with data packets, congestion occurs and unicast control packets have to compete with the data packets for gaining access to the medium while probability of broadcast control packets colliding with unicast packets increases. In such scenarios, data packets can get dropped and that limits the throughput. Furthermore, because data packets have to compete with routing packets,



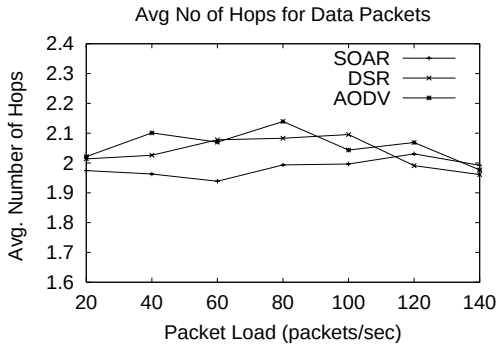
(a) Number of control packets produced



(b) Fraction of data packets received



(c) Average delay of data packets



(d) Average number of hops traversed

Figure 2.14: Effect of loading 20-node network (number of sources = 10)

the delay of data packets should be higher.

We analyzed the performance of the routing protocols when the network becomes more congested. For a 20-node network with 10 flows and *pause time* equal to 0 s, we have used packet loads of 20, 40, 60, 80, 100, 120, and 140 packets/s.

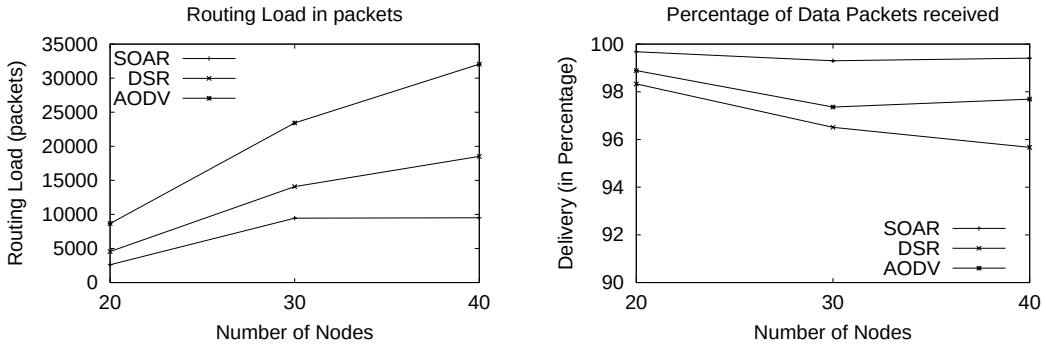
The average delay of data packets in all protocols increases when the load increases due to the increasing waiting time in the interface queue (Fig. 2.14(c)). In all cases, the average delay in AODV is shorter than in DSR and SOAR because data packets in SOAR or DSR suffer more from queueing delay.

We see from Fig. 2.14(a) that control overhead is always less in SOAR than in

DSR and AODV. We also observe that on heavier load, SOAR and AODV experience higher rate of increase of control packet overhead than DSR, primarily because SOAR and AODV's bulk of control packets are broadcast, which can collide with unicast data packets while DSR's majority of control packets are unicast replies.

From Fig. 2.14 we see that, in terms of data delivery, SOAR's performance is always better than DSR and AODV when the traffic load is less than 120 packets/s. At a packet load higher than 100 packets/s, AODV performs better than SOAR or DSR, because SOAR and DSR have stale link-state information, leading to more drops under heavy loads at forwarding nodes.

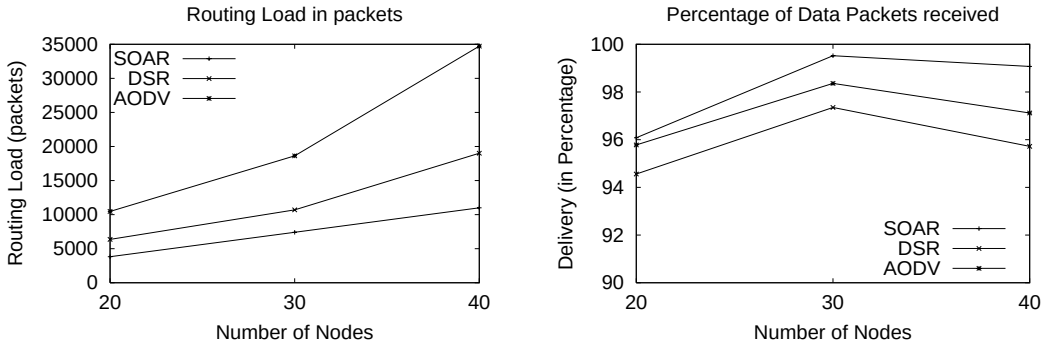
2.3.5 Effect of Network Size



(a) Number of control packets produced (b) Percentage of data packets received

Figure 2.15: Effect of network size (pause time = 0 s)

We have simulated the effect of network size on the performance of each of the routing protocols (Fig. 2.15, Fig. 2.16). We have increased the size of network from 20 to 30 to 40 with the total load being kept constant at 40 packets/s. We have proportionally increased the area of coverage from 1000m×300m to 1000m×500m, to



(a) Number of control packets produced (b) Percentage of data packets received

Figure 2.16: Effect of network size (pause time = 60 s)

1000m×700m to keep the network density similar. At the start of the simulation, the optimum path length is higher with bigger network size; however, due to randomized mobility, the average path length has remained almost same for high mobility scenarios. Here our approach is to test how the different protocols behave with higher number of nodes while maintaining paths for same 50% of the total nodes of flows i.e. 10 flows for 20 nodes, 15 flows for 30 nodes and 20 flows for 40 nodes.

From Fig. 2.15(a) and Fig. 2.16(a) we see that the increase in control packets is similar for all protocols when the network size increases from 20 to 30, but the increase is more pronounced for both DSR and AODV when the number of nodes increases further to 40, while it remains the same in SOAR. This is true for both the cases with pause time equal to 0 second and 60 seconds. One of the reasons behind this is SOAR produces fewer queries ($< 10\%$) compared to DSR (33%) or AODV (80%), which become costly when the size of the network increases. For SOAR due to high amount of redundancy, most of the time it has alternate paths. If the paths are stale then updates need to be exchanged which involve fewer nodes than the network wide

queries. Observe that in terms of data packet delivery, (Fig. 2.15(b), Fig. 2.16(b)) SOAR performs always better than DSR or AODV and the difference increases with the increase of the number of nodes. For these high mobility cases (pause time equal to 0 seconds and 60 seconds) AODV delivers more than DSR for all network sizes, though in terms of control packet overhead DSR is always better. If byte overhead in routing packets is concerned DSR performs slightly better than SOAR for all cases while AODV's byte overhead is always higher than SOAR and DSR. Though in general SOAR sends larger sized control packets, the small number of control packets it sends reduces drastically the byte overhead.

2.4 Conclusions

We have presented, verified and analyzed SOAR, the first on-demand routing protocol based on source-tree information represented with link-states.

Our study shows that SOAR is an effective protocol for mobile ad-hoc networks, and its approach to route reporting and route maintenance is more efficient than DSR nd AODV, which are representatives of the state-of-the-art on-demand routing protocols for mobile ad-hoc networks. The performance improvement obtained in SOAR compared to DSR and AODV is a direct result of communicating source trees rather than distances or paths to active destinations in route requests and route replies. Exchanging source trees provides high redundancy, which reduces the frequency with which route discoveries are needed.

Chapter 3

Path Selection for On Demand Link-State Routing

While choosing routes for any destination in on-demand link-state routing protocols, apart from the shortest path criterion an additional constraint that only selected neighbors can be selected as next hops to reach certain destinations has to be satisfied. To elucidate, a situation can arise at a router where based on its view of the network topology, some nodes can be reached via the shortest paths through certain neighbors. However, it can happen that those neighbors have not advertised routes to those destinations and therefore, they might not know of any route to those destinations and packets forwarded to them would be dropped. Therefore, to prevent incorrect data packet forwarding, longer paths have to be selected for those destinations. This constraint motivates a new path selection selection algorithm because the traditional path-selection algorithms like the Dijkstra's algorithm or the Bellman-Ford algorithm

computes shortest paths based on only a single metric like hop count, delay or bandwidth.

Section 3.1 describes the constraint in the path selection algorithm for on-demand link-state routing and shows how a modification to the basic Bellman-Ford Algorithm forms a source graph rather than a source tree while it satisfies the constraint of the path-selection. Section 3.2 shows why the computation of a source tree with the given constraint is an NP complete problem and describes a polynomial-time approximating solution. Section 3.4 shows how source-tree based routing combined with this path selection algorithm can be applied for policy-based routing. Section 3.5 concludes the chapter.

3.1 Building a Source Graph

We have seen in the previous chapter that unlike previous pro-active link-state protocols, SOAR advertises paths to only *important* nodes. This can lead to a situation in which a node i finds that neighbor k should be on the shortest path to a destination j, but node k has not advertised any path to node j. The reason can be that node k does not have a route for node j. To prevent packet losses, the path selection algorithm should ensure that the links in the anticipated path from node i to node j through neighbor k has been advertised by neighbor k itself.

We illustrate this scenario using Fig. 3.1, which shows the partial topology at node a. The partial topology at node a has been computed based on the inputs from two neighbors b and c. Let us assume that node h is important for nodes $a,\ b,\ e,\ f,$

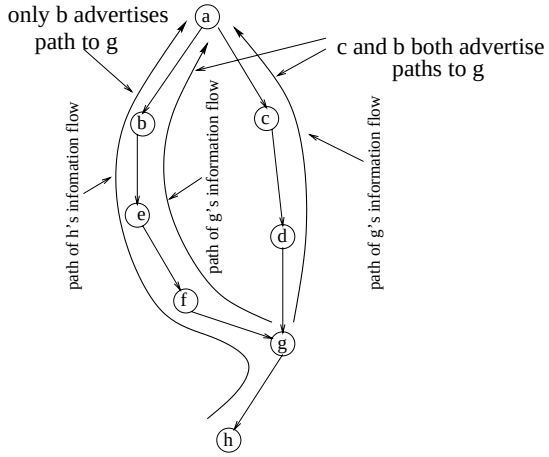


Figure 3.1: Constraints in path selection for on-demand link-state routing

and g, while node h is not important for node c and node d. Assume also that node g is important for all nodes. Because node h is not important for node d, propagation of path information for node h stops at node d and node a does not hear from node c about node h, although node c should be in the shortest path from node a to node h. Node a learns about link (g, h) from node b.

Given a directed graph, a traditional path selection algorithm like the Dijkstra's shortest path first algorithm or the Bellman-Ford algorithm would find the shortest paths from a source to any destination on the basis of a single metric. When any of these algorithms is run on the partial topology at node a, node c will be selected as the next hop to reach node g and node h. Unfortunately, because node c does not know about node $h's$ existence, data packets for node h forwarded to node c will be dropped. That shows that the Dijkstra's algorithm or the Bellman-Ford algorithm is inappropriate for computing routes under these circumstances.

Therefore, the above constraint leads to the following property that a path selection algorithm should satisfy in order to start correct packet forwarding:

Property A: *All links in the computed path to a destination through a neighbor should be advertised by the neighbor itself.*

Essentially, the combination of paths used for packet forwarding can form a graph. However, in SOAR certain links in the computed source graph are excluded and a source tree is always advertised with additional information to take care of the difference. For example, at node a, the source tree that would be advertised by node a would be $acdgh$, along with the information that neighbor b is the actual next hop for node h rather than node c. The last information will enable the recipient of the source tree to get a partial view of the actual forwarding graph of the sender, though the complete path would still be not known. Sending trees is preferred to sending to sending graphs to save bandwidth and power.

3.1.1 Modified Bellman-Ford Algorithm

First we describe the new path selection algorithm that has been adapted from the Bellman-Ford algorithm.

The Bellman-Ford algorithm does not consider any constraint apart from choosing the shortest path for any destination. However, a slight modification to the basic algorithm can compute the shortest path for any destination that is also valid with a constraint by exploiting the fact that the Bellman-Ford algorithm explores all paths of length $(|V| - 1)$ to any destination and chooses the shortest among them.

Table 3.1: Terminology used for the path selection process

Modified Bellman-Ford Algorithm		
$G(V, E)$	:	network topology with vertex set, V and edge set, E
NL_s	:	list of neighbors of node s
n_x	:	any neighbor of node s
$d(u, n_x)$	:	distance to node u through neighbor n_x of node s
$\pi(u, n_x)$	:	parent of node u if the path is through neighbor n_x
$d(u)$	:	distance in the shortest valid path to node u
$\pi(u)$	:	parent in the shortest valid path to node u
$next\_hop(u)$	:	neighbor of node s through which node u can be reached
ST_{n_x}	:	source tree reported by neighbor n_x to node s
DFS based Path Selection		
min_dist	:	minimum length of the best paths to a destination x
best_options	:	total number of neighbors that have advertised paths of smallest length
c_x	:	nodes which are directly connected to node x and reachable along the least-cost paths through node x
bnh (best next hop)	:	neighbors corresponding to least-cost paths to node x
p_x	:	predecessors of node x in the least-cost paths through which node x can be reached
selec_best_options	:	number of best choices among the shortest paths
count	:	total number of nodes farther from a node that can be potentially included in the final source tree when *nh* is chosen as next hop)

The change that can be incorporated is that all the paths rather than the shortest path can be stored corresponding to each node and only those which satisfy Property A can be considered for shortest route computation.

The input to the algorithm is a graph consisting of nodes and edges. Each edge in the graph has the list of neighbors who have advertised it. There is a change in the data structure maintained for every node in the modified Bellman-Ford algorithm compared to that in the basic Bellman-Ford algorithm. Instead of keeping only shortest distance entry ($d(u)$) for a node u, this algorithm maintains $d(u, n_x)$ entry $\forall n_x \in NL_s$

entry. (For terminology refer to Table 3.1.1). In Procedure Initialize (initialization phase of the algorithm), distance to any node through any neighbor n_x is made infinite and the parent is set to NULL while the distance for the source is made zero and the parent set to itself.

```
Procedure Initialize (G,s)
1  foreach u ∈ V[G]
2      foreach n_x ∈ NL_s
3          d(u, n_x) ← ∞
4          π(u, n_x) ← NULL
5      end
6  end
7  foreach n_x ∈ NL_s
8      d(s, n_x) ← 0
9      π(s, n_x) ← s
10 end
```

In Procedure Mod_Bellman_Ford, steps 5-11, iterate through each vertex in G, and then through all the outgoing links of each vertex in G. Each edge visited is used for relaxation of path costs (Step 9). Steps 4-13 will be executed, until no change in $d(u, n_x)$ for any node u for any neighbor n_x occurs in an iteration. This helps to terminate the algorithm earlier, without requiring the total number of iterations to be always $(|V| - 1)$, which is required in the worst case. The steps in the Procedure Mod_Bellman_Ford function are similar to the steps of the original Bellman-Ford algorithm, the difference being mainly in the Relax procedure called in Step 9.

```
Procedure Mod_Bellman_Ford (G, s)
1  Initialize (G, s)
2  any_change ← TRUE
3  while (any_change ≠ FALSE)
4      any_change ← FALSE
5      for (i → 1 to |V[G]|
6          u ← node_i
7          do foreach edge (u,*)
8              if ((u, v) ≠ (u, s) AND l_u^v.cost ≠ ∞)
9                  any_change ← any_change∪ Relax(u, v)
```

```
10              endif
11          end
12      end
13  end
```

In Procedure Relax, steps 3-8, for each neighbor n_x of NL_s, who has advertised the link l_u^v, $d(v, n_x)$ is relaxed. Step 3 ensures that a path to v through n_x should be advertised by n_x.

The procedures described so far combine to compute all the shortest valid paths, where each path satisfies Property A. In case multiple shortest valid paths are possible for any destination and one of them already exists in the current forwarding structure, then that path is chosen otherwise any random choice would suffice. The combination of the shortest valid path for each destination will form a source graph. Then the source tree to be advertised is formed by combining edges connecting each node to its parent. If a path is selected for any destination, then the immediate predecessor in that path becomes the parent of that destination. Procedure makeSrcTree builds the final source tree.

```
    Procedure Relax (u,v)
1   any_change ← FALSE
2   foreach n_x ∈ NL_s
3       if (l_u^v ∈ ST_{n_x})
4           if (d(v, n_x) > d(u, n_x) + l_u^v.cost)
5               d(v, n_x) = d(u, n_x) + l_u^v.cost
6               π(v, n_x) = u
7               any_change ← TRUE
8           endif
9       endif
10  end
```

```
    Procedure makeSrcTree
1   d(v, ∗) = min(d(v, n_x) ∀n_x ∈ NL_s;
2   ST_i ← NULL
3   foreach v ∈ NL_s
```

```
4       ST_i ← ST_i ∪ {s, v}
5   end
6   for (i = 2; i ≤ max_dest; i++)
7       foreach v with d(v, *) = i
8           foreach n_x with d(v, n_x) = i
9               v.nh ← n_x
10                v.parent ← π(v, n_x)
11                if (nh = RT_i.v.next_hop)
12                        break
13                endif
14            end
15        end
16  end
17  foreach v ∈ V
18      ST_i ← ST_i ∪ {v.parent, v}
19  ————————————————————————————————————
```

The complexity of modified Bellman-Ford Algorithm is $O(n^2d^2)$, where n is the number of nodes in the network and d is the average degree of each node. In comparison, the unmodified Bellman-Ford algorithm has complexity of $O(n^2d)$. Intuitively the difference is due to the storage of d paths instead of a single one in the modified Bellman-Ford algorithm.

3.1.2 Forced Routing

One important observation from the example of Fig. 3.1 is that if some neighbors would have advertised paths for certain destinations, then shorter paths can be chosen for those destinations and the advertised source tree would then be exactly same as the forwarding tree of each node. Forced routing is used to make that happen. Referring to the example of Fig. 3.1, node a knows of a shorter path to node h through node c, but it cannot use node c as the next hop for node h because it has not reported it. Therefore, forced routing would be used to ensure that neighbor c advertises a path for node h also.

For the scenario in Fig. 3.1, node a sends a *ForcedUpdate* towards node h

through neighbor c such that all intermediate nodes c and d along the shortest path to node h start considering node h as important, and *ForcedReplies* containing the shortest path information retraces from node d and node c to node a. For the brief time, before shorter routes get established using forced routing, data packets will be forwarded along the longer paths.

ForcedUpdates are also used by a router in SOAR to correct neighbors when they send invalid path information for any destination in reply to its queries. Unlike queries, replies or updates, which are broadcast packets sent to all neighbors, ForcedUpdates and ForcedReplies are unicast packets sent to specific neighbors.

3.2 Building a Source Tree

Referring back to the example of Fig. 3.1, we can see that for a certain time interval when forced routing gets accomplished and packet forwarding continues along the shortest path through node b, the source tree advertised by node a does not truly advertise the true picture of its forwarding graph or source graph. That can lead to loops. In order to prevent that situation, next we need to develop a new path-selection algorithm that computes a source tree, rather than graph, while satisfying the path constraints of on-demand link-state routing.

The constraint imposed in computing the source tree can be translated to the following rules:

Rule 1: If p_i^{kj} is the path computed at node i to reach node j through neighbor k, then for each edge $e \in p_i^{kj}$, node k belongs to $lab(e)$ where $lab(e)$ indicates the label

set for edge e. The label set refers to the neighbors which have advertised that edge. Hereafter, the words label and neighbor have been used interchangeably.

Rule 2: There can be several potential paths that satisfy Rule 1. However, for final route computation, paths with the smallest length should be chosen.

Rule 3: If a particular node can be reached via multiple paths due to Rule 1 and Rule 2, only a single path can be chosen to reach that node such that a source tree is formed.

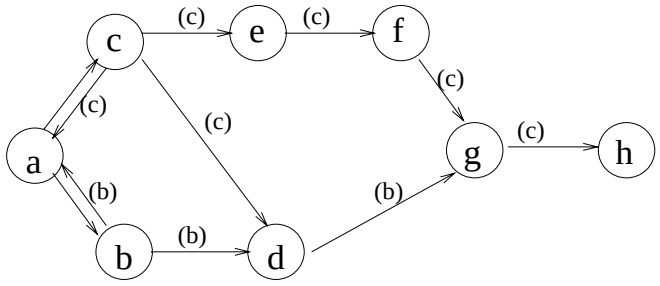


Figure 3.2: Network topology at node a based on inputs from neighbors b and c. Each link lists neighbors who have advertised that link.

For example, in Fig. 3.2 which shows the partial topology at node a and the labels corresponding to each link, there are two valid paths for node g namely $abdg$ and $acefg$. Path $abdg$ through neighbor b will be selected because that is the shortest path (Rule 2). In that case, no finite-cost path is possible for node h because the only valid path $acefgh$ to reach node h is possible through neighbor c (Rule 1) and the shortest valid path to node g, the predecessor according to that path is through node b.

3.2.1 Complexity Analysis

As shown in the previous example, finite-cost paths for all destinations might not be possible. In such a case, an optimal path-selection algorithm has to be developed that maximizes the number of nodes for which finite-cost paths can be obtained. This problem of finding the source tree with the maximum number of vertices is termed as OPT-TREE problem and is defined as follows:

INSTANCE : Graph, G = (V, E), a given node i, which can be termed as the source ($i \in V$) and a collection of possible labels, C, and the set of labels, S_e corresponding to each edge e ($S_e \subseteq C$, $e \in E$).

The labels corresponding to any edge can be thought of as boolean variables and each variable would be assigned an '1' if the edge is included in the final source tree and the path containing that edge includes the neighbor that has been represented by that variable. Therefore, for the link e with label set, S_e $\{l_1, l_2, l_3\}$, the boolean value assignment would be $\{l_1 = 1, l_2 = 0, l_3 = 0\}$, if the edge e is included in a path of the final source tree through neighbor l_1 . If $l_4 \in C$, but $l_4 \notin S_e$, then l_4 would be undefined for S_e.

Based on the above boolean assignment strategy, the OPT-TREE problem can be re-defined as :

QUESTION : What is the maximum-vertex source tree that can be formed at node i, such that the edges on the path from node i to any node v in that source tree will have the same label set to "1" ?

We explain OPT-TREE problem in detail using Fig 3.3. Fig 3.3(a) shows

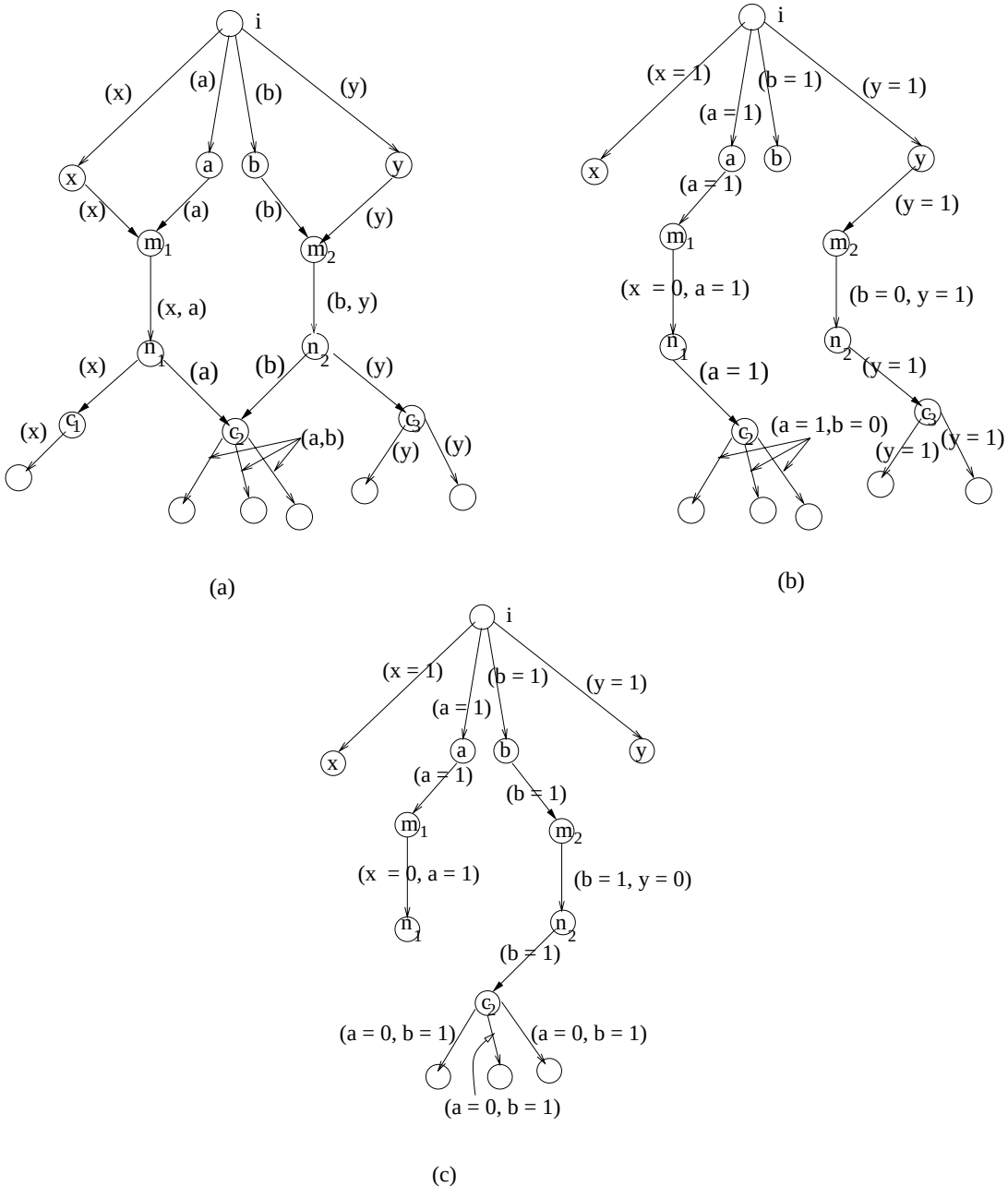


Figure 3.3: (a) Partial topology at node i (b) Optimal source tree and (c) A source tree satisfying the rules of policy constrained path selection

an instance of the *OPT-TREE* problem, i.e., the network topology at node i and the labels associated with each edge in the topology. Fig 3.3(b) gives the optimal solution that satisfies all the rules for constructing the source tree. As shown in Fig. 3.3(b), all edges from node i to node n_1 has the same label $\{a\}$ set to "1", while all edges from node i to node n_2 will have the same label $\{y\}$ set to "1". There are also other non-optimal solutions available satisfying all relevant rules and one of them is as shown in Fig. 3.3(c).

The next step towards solving the OPT-TREE problem is to reframe the OPT-TREE problem as a k-TREE problem: *Is there a source tree having k vertices and rooted at source i, such that edges on the path from node i to any vertex v will have the same label set to "1"?* By varying the value of k from one to $|V|$ and applying the k-TREE problem, we have the solution for the optimality problem. We will show that the k-TREE problem is NP-complete. And subsequently OPT-TREE will also be NP-complete.

As the first step towards proving that *k-TREE* problem is NP-complete, we define the *SPAN-TREE* problem.

INSTANCE: Graph, G = (V, E), a given node i, which can be termed as the source, a collection of labels C, and the set of labels, S_e corresponding to each edge e ($S_e \subseteq C$).

QUESTION: Can we construct a tree rooted at node i containing $|V|$ nodes, such that edges on the path from node i to any node v have the same label set to "1"?

We first show that SPAN-TREE $\in NP$. Let the tree formed be $G = (V', E')$.

We first verify $V' = V$ and then we verify that in the spanning tree formed, all the edges in the path from the source to any node will have the same label set to "1" and this can done in O($|V|$)) time.

The next step is to prove that SPAN-TREE is NP-hard. For that purpose, the well-known 3SAT problem [6, 13] has been reduced to SPAN-TREE. In the well-known 3SAT problem, the question is to find whether a logical formula in the 3-CNF form is satisfiable or not.[1] An example of a 3SAT problem is how to assign values to the literals, x_1, x_2, and x_3 such that the logical formula $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$ is satisfiable, i.e., has an output equal to "1".

The reduction algorithm consists of three steps. The first step is to convert the logical formula $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$ to the following form :

$(x_1 \vee \neg x_1 \vee y) \wedge (x_2 \vee \neg x_2 \vee y) \wedge (x_3 \vee \neg x_3 \vee y) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$, where y is a dummy literal which always assumes the boolean value zero.

The modified formula is essentially the same as the original formula because the first three clauses are true, irrespective of the values set for x_1, x_2, and x_3. This step of reduction is trivial, because it extends a logical formula by adding a clause corresponding to each literal $x \in C$, where each clause is $(x \vee \neg x \vee 0)$ and C is the set of all literals. The first step of the reduction can be done in $O(m)$ time, where m is the cardinality of the set C.

[1] A boolean formula is in 3-CNF (conjunctive normal form) if is expressed as an AND of clauses, each of which is the OR of three distinct literals)

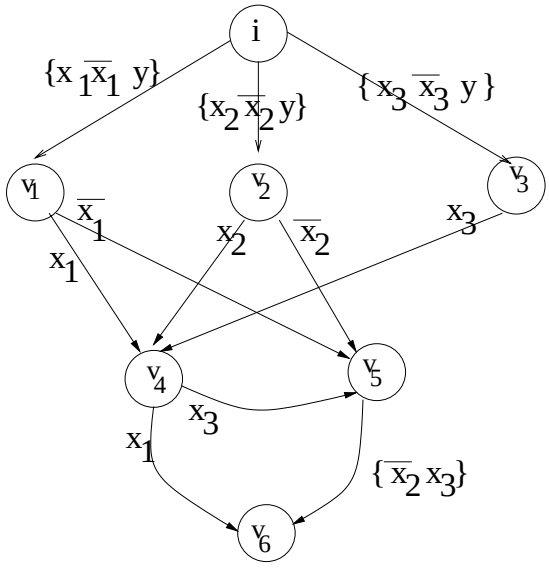


Figure 3.4: Representation of the logical formula $(x_1 \vee \neg x_1 \vee y) \wedge (x_2 \vee \neg x_2 \vee y) \wedge (x_3 \vee \neg x_3 \vee y) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$ during second step for the reduction of 3SAT problem to a SPAN-TREE problem.

The modified logical formula is next represented in the form of a graph. First, we create a node i that represents the root and a new node is created for representing each of the clauses in the modified logical formula. Logically, for each node representing a clause, there are three incident edges and each edge has a label corresponding to each literal in that clause. Because the same literal appears in different clauses, in the graph created there can be multiple edges with the same label set.

The graph is created stepwise by starting from the first clause and moving towards the right of the logical formula and representing each clause by a node. For each literal x_i, the last node v_{last} with an incident edge having label x_i, is remembered. If a new node v_{new} is created for a new clause, one of whose literals is again x_i, an edge from v_{last} to the new node v_{new} is created and is assigned the label x_i. Also v_{new} becomes v_{last}. This process makes the distance of node v_{new} from node i always

greater than the distance of node v_{last} from node i if the path is traced from node i to node v_{new} along the edges with label x_i.

Using the above algorithm, the graph shown in Fig. 3.4 has been created corresponding to the modified logical formula. Nodes v_1, v_2 and v_3 are joined to the source i by edges and they correspond to the new clauses that have been added to the original logical formula. Node v_4 is created for the fourth clause of the modified formula with three incident edges having labels x_1, x_2 and x_3 respectively. Then node v_5 is created, v_5 joins node v_4 by an edge with label x_3 because node v_4 is the last node that has been created with an incident edge having a label x_3. Similarly all the other edges and nodes are created. This representation can be done in polynomial time by constructing $(m+n)$ nodes and $(3n+m)$ edges where m is the number of literals and n is the number of clauses in the original 3-CNF formula.

The next step of the reduction algorithm is to replace each node (except those nodes directly connected to the source) in the graph created above by three nodes each with three incident edges, each with a particular set of labels. This step is shown in Fig. 3.5, where a node with three incident edges with labels x_1, x_2 and x_3 respectively is replaced by three nodes with the labels of the incident edges on each of the three nodes being $[\{\neg x_1\}, \{a\}, \{b\}]$, $[\{x_2\}, \{b\}, \{c\}]$ and $[\{\neg x_3\}, \{c\}, \{d\}]$ respectively. This reduction is similar to the reduction algorithm used for the reduction of the 3SAT problem to the 1in3SAT problem [49].[2] The newly created edges with labels a, b, c, and d directly connect the source to the newly created nodes. Rest of the connections

[2]The problem in 1in3SAt problem is whether a logical formula in the 3-CNF form is satisfiable or not where exactly one literal in a clause is true.

are in the same pattern as shown in Fig. 3.4. In Fig. 3.6, we show how the node v_4 with label set $[x_1 = v_1, x_2 = v_2, x_3 = v_3]$ is replaced by three nodes, v_4^1, v_4^2 and v_4^3 and their label sets are respectively $[\{\neg v_1\}, \{a^1\}, \{b^1\}]$, $[\{v_2\}, \{b^1\}, \{c^1\}]$ and $[\{\neg v_3\}, \{c^1\}, \{d^1\}]$. Only the representation for node v_4 has been shown in the figure for clarity. Each of the nodes, v_5 and v_6 is similarly replaced by three nodes, each with three incident edges.

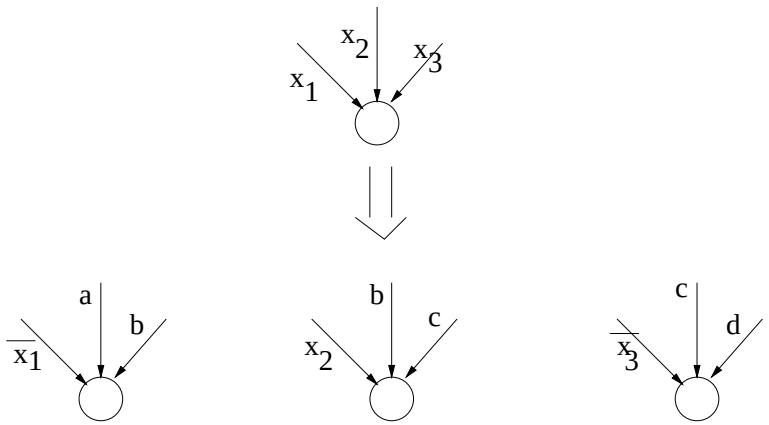


Figure 3.5: Final step in the representation of 3CNF formula in the form of nodes and edges

The next step is to show that there exists a solution in 3SAT problem if and only if there is a solution in SPAN-TREE. First we show that if there exists a solution for 3SAT there will also be a solution in SPAN-TREE. The steps of the proofs presented here have been motivated by the steps given of [49].

If the 3SAT problem has a solution, then the logical formula is satisfiable and each clause will be true and at least one literal of each clause must be true. We now show that depending on the values of x_1, x_2, and x_3 for a clause, $(x_1 \vee x_2 \vee x_3)$, we can have different boolean assignments for labels, a, b, c and d, based on which if there exists a solution for the 3SAT problem, we can have a solution for SPAN-TREE

problem.

Let us assume that $x_2 = 1$ in the clause $\{x_1, x_2, x_3\}$. In that case, following assignment for the labels a, b, c, and d leads to a solution for the SPAN-TREE problem: $a = x_1$, $b = 0$, $c = 0$, $d = x_3$. Using such an assignment, the nodes representing clauses $(\neg x_1 \vee a \vee b)$, $(x_2 \vee b \vee c)$ and $(\neg x_3 \vee c \vee d)$ will have exactly one incident edge having one label set to "1". Therefore, by using appropriate values for a, b, c and d, if there exists a solution for 3SAT, every node in the final graph has exactly one incident edge whose exactly one label has been set to "1", which implies that a spanning tree can be formed. Given that, there is exactly one incident edge on a vertex with exactly one label set to "1", there can be no cycle, because in that case for at least one node (excluding the source which has no incident edge), among all labels for all its incident edges there have to be at least two labels which are set to "1". However, that cannot be true. Moreover, a forest can not be formed because apart from the source node i, each node represents a clause of a satisfiable logical formula and therefore, each node must have exactly one incident edge with exactly one label set to "1".

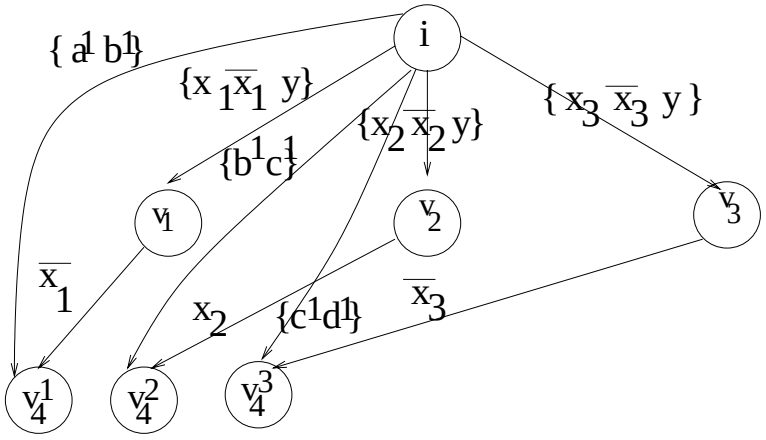


Figure 3.6: Final representation of the clause $(x_1 \vee x_2 \vee x_3)$ in the form of vertices and edges

Let us assume that $x_2 = 0$. In that case there are three possibilities for the values assigned to x_1 and x_3 and they are (a) $x_1 = 1$ and $x_3 = 1$ (b) $x_1 = 1$ and $x_3 = 0$ and (c) $x_1 = 0$ and $x_3 = 1$.

For building a spanning-tree, condition (a) leads to the assignment : a = c = 0, d = 1, b = 1; condition (b) leads to the assignment: b = 1, a = c = d = 0; and condition (c) leads to the assignment: a = b = d = 0, c = 1.

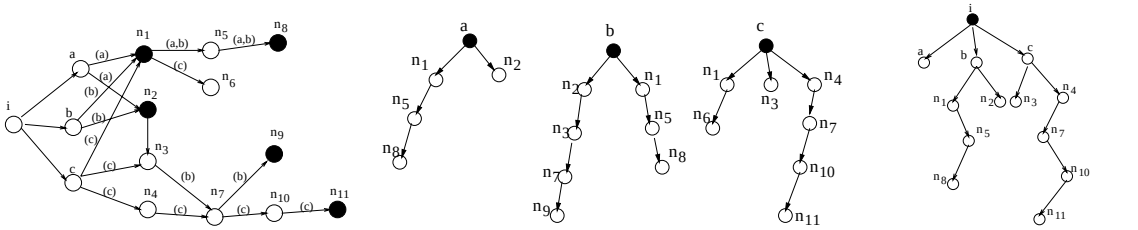
The case of ($x_1 = 0$ and $x_3 = 0$) is not possible, because at least one literal in each clause has to be true.

The next step is to show that, if there exists a solution for the SPAN-TREE problem, there exists also a solution for the 3SAT problem. The proof is by contradiction. Let us assume that, although there exists a solution to the SPAN-TREE problem, there exists no solution for the 3SAT problem. If such is the case, then $x_1 = x_2 = x_3 = 0$ for at least one clause. Then $a = 0, d = 0$. Also $b = 0$ or $c = 0$ (but not both b and c). However, if either one of these two literals, b or c is equal to one, then the condition that in the spanning tree there is only one incident edge with exactly one label set to "1" is not satisfied. For the first node (that represents the clause $(\neg x_1 \vee a \vee b)$) or the third node (that represents the clause $(x_2 \vee b \vee c)$) we will get two incident edges that will have labels set to "1", which is a contradiction to the original assumption that a spanning tree has been formed in which case each node can have exactly one label set to "1" among all labels for all of its incident edges. Therefore, the 3SAT problem has a solution, if SPAN-TREE has a solution.

We have thus shown that the SPAN-TREE problem is NP-hard and that

SPAN-TREE $\in$ NP, which implies that SPAN-TREE is NP-complete.

Now, the next step is to show whether the k-TREE problem is NP-complete. This can be a proof by restriction [13]. The SPAN-TREE problem is a restricted case of k-TREE problem, where the value of k is $|V|$. Therefore, k-TREE is NP-complete and also the OPT-TREE problem, i.e., the problem of finding the tree with the optimal number of nodes in it is an NP-complete problem, because the decision problem (k-TREE) to which the optimality problem has been reframed is NP-complete.



(a) Partial topology (b) Source graphs corresponding to neighbors a, b, c at node i (c) Final source tree

Figure 3.7: Inputs and outputs of path selection algorithm

3.2.2 Finding Valid Paths

We have shown in the previous section that the OPT-TREE problem is NP-complete. That implies no polynomial-time optimal solution is possible. Therefore we have designed a polynomial-time approximating solution that satisfies the given constraints. The heuristic consists of selecting the valid paths for a destination and then choosing the shortest among the valid paths with the objective of having maximum nodes in the final source tree. Next we describe each phase of the path selection algorithm and Table 3.2 explains the steps through examples.

The inputs for the path selection algorithm are the source trees advertised by each neighbor, n_i ($n_i \in N_i$, where N_i is the set of neighbors of node i and $d = |N_i|$). Fig. 3.7(a) shows the partial topology at node i, where the list of labels for each link is shown in parenthesis next to the link. Fig. 3.7(b) shows the source trees of neighbors a, b and c that are stored at node i.

Any path to a destination that is in the source tree of a neighbor is a valid path. By executing Depth First Search (DFS) on each source tree, a valid route for each destination in the form of tuple {distance, successor, predecessor} is computed. Atmost d valid paths are theoretically possible for each destination, and the actual number depends on the maximum number of source trees of neighbors in which a path to the destination exists. The complexity of depth-first traversal is $O(n)$ for a tree with n nodes; therefore, the total complexity is $O(nd)$ for d neighbors. Column 2 of Table 3.2 shows the different possible valid paths in the form of $\{nexthop, predecessor, distance\}$ tuples for each node in the network of Fig. 3.7. For example, corresponding to node n_8 the valid paths are: (1) through neighbor a, with predecessor n_5 and distance four (this path is represented by the tuple $\{a, n_5, 4\}$), and (2) through neighbor b, with predecessor n_5 and distance four (this path is represented by the tuple $\{b, n_5, 4\}$).

3.2.3 Choosing the Best Paths

After computing all valid paths for a particular destination, the least-cost paths among them are only considered for the final route selection, thereby satisfying Rule 2. That process requires two operations: (a) finding the minimum cost among the

possible options ($O(nd)$), and (b) selecting the paths which are of least cost ($O(nd)$). Columns 5, 6, and 7 of Table 3.2 show the valid least-cost paths possible for each destination in the form of its direct children, successor and predecessor respectively. A valid path to any destination will have the same label as the label of the links that form the path. As shown in Table 3.2, the least-cost valid path to reach node n_7 among the two valid paths with labels b and c has label c, with the predecessor in that path being node n_4 and the possible direct children being nodes n_9 and n_{10}.

The next step of the operation is to choose a single path for each destination among the valid least-cost paths, aggregation of which forms the maximum-vertex source tree. We have already discussed that this problem is NP-complete. The solution that we will provide is a polynomial time approximating solution. The steps of the proposed solution are given in Fig. 3.8. As shown in Fig. 3.8, imagine that the nodes are all arranged in several layers, with source i at the highest layer and the nodes with the longest least-cost paths at the bottom-most layer. Based on the successor-predecessor relationship of all the valid least-cost paths, the nodes are then joined with one another. This will form the input to the heuristic and the heuristic is divided into two separate operations. In the first operation, calculations are made from the nodes farthest from node i to the ones nearest. If a particular edge with a certain label is chosen to be the incident link on a node then under such a label assignment the number of downstream nodes or children further down the layer that can be added is counted. The links with labels giving maximum count at a node are ultimately considered for inclusion in the final source tree. To illustrate, if node n_1 is reached

Table 3.2: Step-wise execution of the DFS-based path selection algorithm

Node	{next_hop, predecessor, distance}	min _dist	best _options	c_x	bnh	p_x	[selec_best_options, count, {nh, dist}]	[nh, pred, dist]
a	{a, i, 1}	1	1	n_1, n_1	a	i	X	[a, a, 1]
b	{b, i, 1}	1	1	n_1, n_2	b	i	X	[b, b, 1]
c	{c, i, 1}	1	1	n_1, n_3, n_4	c	i	X	[c, c, 1]
n_1	{a, a, 2}, {b, b, 2}, {c, c, 2}	2	3	n_5, n_6	a, b, c	a, b, c	[2, 3, {a, 3}, {b, 3}]	[b, b, 2]
n_2	{a, a, 2}, {b, b, 2}	2	2	X	a, b	a, b	[2, 1, {a, 1}, {b, 1}]	[b, b, 2]
n_3	{b, b, 2}, {c, c, 2}	2	1	X	c	c	[1, 1, {c, 1}]	[c, c, 1]
n_4	{c, c, 2}	2	1	n_7	c	c	[1, 4, {c, 4}]	[c, c, 1]
n_5	{a, n_1, 3}, {b, n_1, 3}	3	2	n_8	a, b	n_1, n_1	[2, 2, {a, 2} {b, 2}]	[b, n_4, 3]
n_6	{c, n_1, 3}	3	1	X	c	n_1	[1, 1, {c, 1}]	[NULL, NULL, ∞]
n_7	{b, n_3, 4}, {c, n_4, 3}	3	1	n_9, n_{10}	c	n_4	[1, 3, {c, 3}]	[c, n_4, 3]
n_8	{a, n_5, 4}, {b, n_5, 4}	4	2	X	a, b	n_5, n_5	[2, 1, {a, 1} {b, 1}]	[b, n_5, 4]
n_9	{b, n_7, 5}	5	1	X	b	n_7	[1, 1, {b, 1}]	[NULL, NULL, ∞]
n_{10}	{c, n_7, 4}	4	1	n_{11}	c	n_7	[1, 2, {c, 2}]	[c, n_7, 4]
n_{11}	{c, n_10, 5}	5	1	X	c	n_{10}	[1, 1, {c, 1}]	[c, n_{10}, 5]

via neighbor c, then nodes which can be reached from node n_1 are n_1 and n_6. While if either node a or node b is chosen as the next hop to reach node n_1 then nodes n_1, n_5 and n_8 can be included. There for the later cases the count is three as opposed to the first case whose count is two. Hence, node c is excluded as a next hop choice for destination n_1 and its total number of possible least-cost paths with maximum count (total number of possible least-cost paths is referred to as *selec_best_options* in Table 3.2) becomes two and they have labels a and b.

During the second operation, the final selection of a successor for each destination among *selec_best_options* is done stepwise starting from the nodes nearest to node i continuing till the farthest nodes. Final selection of next hop is made based on the following criteria: (1) if the present successor is among the *selec_best_options*, that node will be chosen automatically as the successor to prevent route flapping, (2) if a node has a single predecessor, then it automatically selects the next-hop chosen by the predecessor as its successor and (3) still if there are several choices, random selection is made. For the example of Fig. 3.7, node n_2 can be reached via neighbor a, as well as via neighbor b. Any one of node a or node b can be chosen as *next_hop*. However, if node b is the present *next_hop*, then that becomes the automatic choice. Node n_5's only valid predecessor is node n_1, therefore, the successor to reach node n_5 is node b. Fig. 3.7(c) shows the minimal source tree drawn from the final results in the last column of Table 3.2.

The complexity of the above two operations is $O(nd + 2nd^2)$. Accordingly, by considering each step taken for the path selection, the complexity of the entire

path selection algorithm becomes $O(nd^2)$ where d is the node density. In comparison, the complexity of Dijkstra's algorithm is $O(n^2)$ which implies the new path selection algorithm will scale well with the number of nodes in the network, when the network density remains constant.

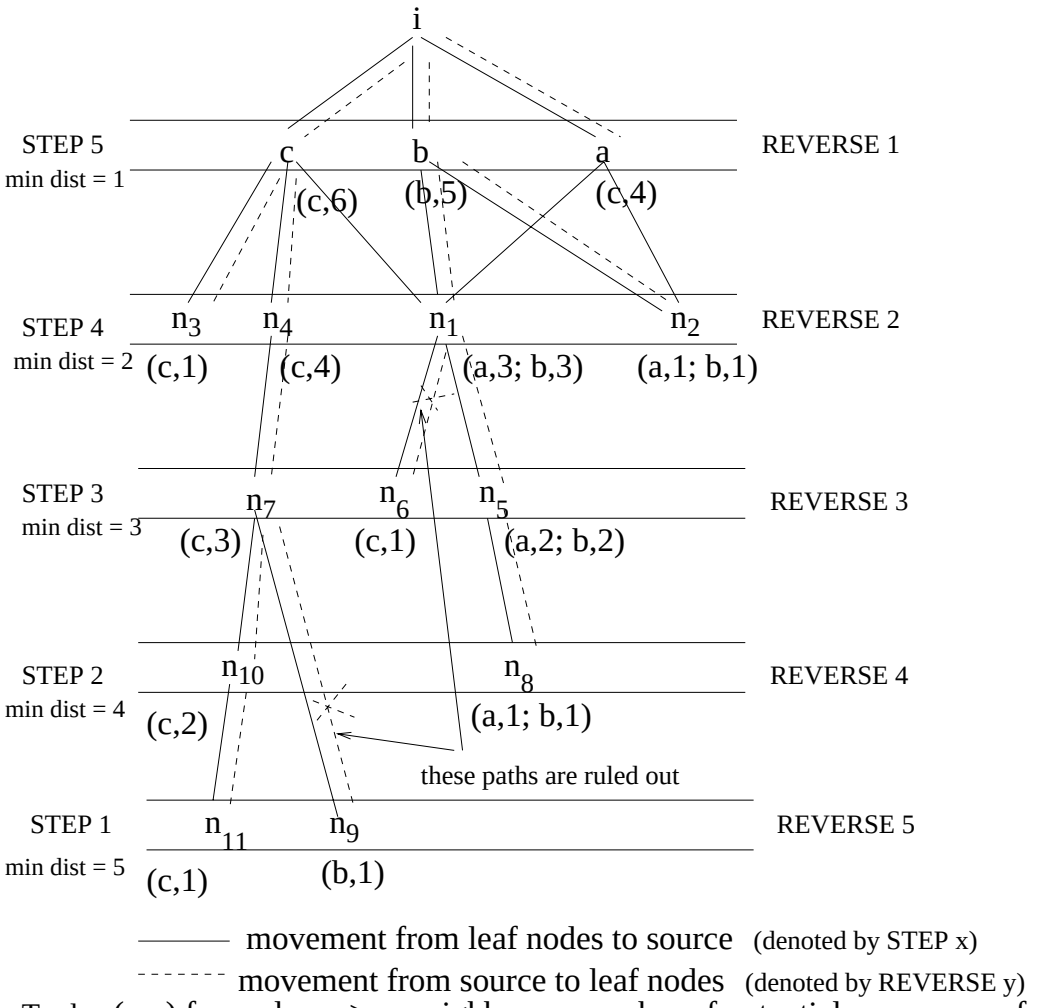


Figure 3.8: Depiction of the last phase of the new path selection algorithm

3.2.4 Optimizing the Source Tree

According to the example of Fig. 3.7, node n_6's only valid path is through neighbor c with predecessor n_1, but node n_1 is reachable through neighbor b in the final source tree. Therefore, node n_6 has to be excluded from the final source tree. If the label of link (n_1, n_6) is b, instead of node c node n_6 can be included in the final source tree. This shows that a change in label can improve the reachability of the nodes.

Next we describe how the changes in the labels can be computed such that the final source tree becomes the optimal source tree. Using the Dijkstra's algorithm without considering any constraint in the choice of the next hop, the minimum distance to reach any node v (d^v_{min}) and the predecessor for that path (π^v_{min}) is first computed. If d^v_{min} is less than the distance d_s of the shortest path taking into consideration the constraint in the choice of next hop, then the label changes for the link incident on node v are computed. Procedure computeLabelChanges shows that process.

Procedure computeLabelChanges

```
1     foreach node v ∈ NL_s
2         v.label = v
3     diameter = max(d^v_min) ∀v ∈ V
4     for( i = 2; i ≤ diameter; i++)
5         foreach v ∈ V if (v.d_min = i)
6             if (d_s > d^v_min)
7                     label of link(v.π^v_min, v) = v.π_min.label ; LABEL CHANGE COMPUTED HERE
8                     v.label = v.π^v_min.label
9
10            else
11                    v.label = label obtained from constraint-based path selection algorithm
12            endif
13        end
14    end
```

3.3 Correctness of the Constrained Path-Selection Algorithm

Theorem 5 *If $ST_i = (V^{'}, E^{'})$ is the final computed source tree, the distance to each node $v \in V^{'}$, according to ST_i is the smallest for node v based on the given topology $G = (V, E)$ under the constraints of Rule 1, Rule 2 and Rule 3.*

Let d_v be the distance to node $v \in V^{'}$ according to ST_i and $\delta^*(i, v)$ be the shortest distance from i to v under the constraints of Rule 1, Rule 2 and Rule 3, given a topology at node i and the set of labels for each edge in the topology. Using depth-first traversals through the source graph, corresponding to each neighbor, we compute $\delta^n(i, v)$, i.e., the distance from i to v through neighbor n.

Let $\quad d_v^n = \delta^n(i, v) \quad \forall n \in N_i$ (N_i : neighbor set).

According to the algorithm, the best next hops ($bnh \in N_i$) for any destination v are chosen such that the following is satisfied,

$$d_v^{bnh} = \min[\delta^n(i, v)] \quad \forall n \in N_i.$$

Because of Rule 1, $d_v^{bnh} = \delta^*(i, v)$ for each bnh. In ST_i, snh is the finally selected next hop for reaching v and snh is selected from the set of bnhs, which implies that $d_v = d_v^{snh} = \delta^*(i, v)$.

Theorem 6 *Using a given topology if one run of the path selection algorithm does not yield optimal solution, the optimal solution can be obtained by message passing with certain relevant nodes.*

Let v be any node that is included in ST_i, i.e., $v \in V^{'}$. Let nh_v be the

next hop to reach v according to ST_i. This implies that $count_v.nh_v = \max[count_v.n_i]$ $\forall\ n_i\ \in\ N_i$, where $count_v.n$ is the total number of nodes in the subtree rooted at v when an edge with label n is selected to be the incident edge on node v.

Assume that a node u has been left out of ST_i (i.e., $u \in (V - V^{'})$). Let an upstream node of u according to one possible least cost path from node i to node u (through neighbor nh_v) be node v and let node v be the last node in the path to node u that belongs to ST_i.

A routing protocol based on link-state information can be defined such that, a node could ask its neighbor(s) to enact a form of *forced routing* along the path [nh_v, ..., v] such that $v, ..., nh_v$ would be forced to advertise to node i the subtree ($SUBT^v$), rooted at node v, containing path to node u and that has been excluded from ST_i.

Let c be the total nodes in $SUBT^v$ excluding v. Then after the exchange of messages, the new count value, $count_v^{'}.nh_v = (count_v.nh_v + c) > \max[count_v.n_i] = \max[count_v^{'}.n_i]$, because forced routing through nh_v increases $count_v.nh_v$ only.

This implies that nh_v would be selected as the next hop for v, and any node u left out in ST_i before would be included, hence giving the optimal solution.

Based on our observations and experimental findings, we have found that sending a source tree is not a good utilization of the available topology information because of the constraint that a single path is only allowed for a node. Advertising and computing source graph would be a more efficient method to approach the problem. We have found that the approximating solution can be for policy based routing in today's Internet.

3.4 Policy-Based Routing Using Path Selection Algorithm

Today's Internet is divided into several autonomous domains and each domain independently defines its own policies based on which routes for destinations are chosen, rather than on the basis of distance-based metrics like delay or hop-count. Border Gateway Protocol (BGP) is an inter-domain routing protocol that is extensively used today and does policy-based routing. However, there is an intrinsic problem in BGP, where certain specifications of policies in different domains can cause BGP to continuously exchange routing messages [47].

Similar situation can arise in secured routing, in which trust relationships between nodes can demand that certain nodes cannot be used for forwarding data to a certain subset of destinations although if physical topology is taken into consideration, those untrusted nodes can provide the shortest paths. Here we present a link-state based routing protocol that addresses the problem of policy-based routing, where routing updates are sent in the form of source trees and computation of source tree is done using the path selection algorithm, presented in the previous section after converting the policies at each domain into labels. Simple Path Vector Protocol (SPVP) has been proposed by Griffin and Wilfong [14] as a form of safe BGP. It suppresses routes automatically when a history attribute corresponding to each route detects a cycle. However, it adopts a reactive approach which takes actions based on past history of route oscillations, while our solution is a proactive approach that ensures that there is convergence under any condition.

3.4.1 Link Vector Protocol

In today's Internet, the well-known existing routing protocols can be broadly classified into two categories : (a) distance-vector protocols (e.g., RIP [16], EIGRP [1], BGP [37]) in which routers exchange vectors of distances of the paths used for all destinations and (b) link-state protocols (e.g., OSPF [28], IS-IS [31]) where a topology is built at each router using flooding of link-state information and each router runs a path-selection algorithm on this topology for final computation of routes. Link-state routing protocols solve the infinite-looping problems of old distance-vector protocols but incur substantial communication overhead [48]. To reduce the huge communication overhead associated with flooding the entire topology in these link-state routing protocols, a new class of distributed routing algorithms has been introduced, e.g., LVA (link-vector algorithm) [3] where links belonging to preferred paths to destinations are exchanged between neighboring routers. The combination of the preferred paths for all destinations forms a source graph. A router builds a partial view of the topology based on adjacent links and the links in the source graphs of its neighbors. Each router runs local path-selection algorithms on its topology to compute its own source graph. A source tree is a subset of a source graph, in which there is only one preferred path for any destination. Because the present-day Internet architecture does not yet support multipath routing and a source-tree exchange mechanism is more bandwidth-efficient, we will assume a source-tree based approach, limiting the number of paths to any destination to one. The Adaptive Link-state Protocol (ALP) [24] is a link-state protocol that uses selective link-state updates like LVA with the contrast that it reduces the

message overhead by not exchanging information regarding explicit deletions of links. STAR (Source Tree Adaptive Routing) [24] is another source-tree based protocol, in which it tries to minimize overhead in lieu of less optimal paths. Link-state updates in STAR are reported to neighbors only if there can be potential loops.

The correctness of the operations of Link Vector Algorithm (LVA) was verified by Behrens and Garcia-Luna-Aceves [3] for any arbitrary type of routing (shortest-path routing, multi-constrained routing or policy routing) under the assumption that a correct and deterministic path-selection algorithm is used and freshness of link-state information can be determined using a mechanism like usage of timestamps. Correctness of a routing protocol implies that it converges within a finite time and it converges to loop-free, correct paths. The correctness proof for LVA can be summarized as follows: following a change in network topology, the head node of the link that undergoes change sends a link-state update. Because the link-state updates can be validated at every node using sequence numbers, no node can continue generating updates indefinitely for any link. Therefore, within a finite time, every node has consistent information about adjacent links and the preferred paths to each of the destinations from each of its neighbors. This information is a subset of the entire topology and is exactly sufficient for computing correct loop-free paths to all destinations.

The correctness proofs of other routing protocols (ALP [26], STAR [24] or SOAR (Chapter 2)) have been presented in literature. The correctness of each of these protocols is based on the assumption that links belonging to preferred paths to destinations are exchanged. The basis of selection of the preferred paths is not relevant

for the correctness of the routing protocol, provided that the path-selection algorithm is correct and deterministic. In order to provide a loop-free solution to the problem of policy-based routing, we advocate the source-tree based routing information exchange. However, we need a unique path selection algorithm that takes into consideration the policy constraints and the path selection algorithm, which has been shown to be correct in the previous section, can serve the purpose. The next step is to show how the policies defined independently at each autonomous system can be represented in the form of labels.

3.4.2 Conversion of Policies to Labels

In policy-constrained inter-domain routing for today's Internet, policies are used to decide (1) which routes to accept from a neighbor (import policy) and (2) the preference with which those routes should be treated (we call it metric policy) and (3) which routes to advertise to neighbors (export policy). The import policy determines how the outgoing traffic is going to be forwarded while the export policy decides how the external traffic enters the domain. The metric policy gives higher preferences to certain paths over other possible paths, such that traffic is mostly sent along those paths.

We are looking at a routing solution, where the gateway routers exchange the routes to network prefixes in the form of source trees. The entire source tree can be sent by a router to its neighbors when routing information is exchanged for the first time between them. Incremental changes to the source trees can next be

advertised. Links are extracted from the source trees of each neighbor to form the topology database at any router.

First we look at the source trees, advertising reachability information for address prefixes and how the import policy would determine which links in the neighbor's source tree would be considered for inclusion in the topology database. Fig. 3.9 shows an example of a source tree, advertised by a router x in Autonomous System x (AS_x) and received at router i.

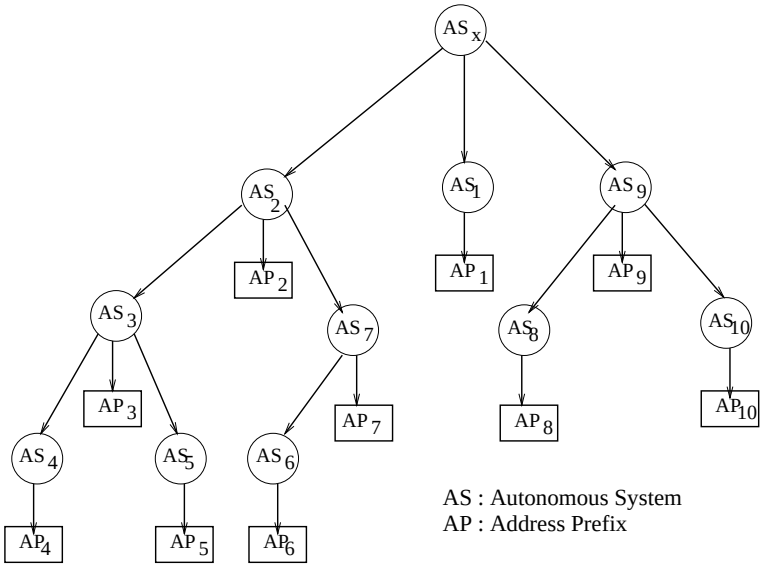


Figure 3.9: Source tree advertised by the router x in autonomous system x (AS_x)

Each node in Fig. 3.9 is either a circle, which represents an autonomous system or a rectangle, which represents an address prefix. An import policy is now used to extract relevant links from these source trees and include them in the final topology. Because the links can be advertised by multiple neighbors, the links need to be validated and they can be done using timestamps, sequence number or by trusting the entry advertised by the neighbor of router i, which is nearest to the head of the

link. Each of the links, which is added to the topology database, is assigned a set of labels, where each label corresponds to the neighbor from which the router has accepted the link-state advertisement.

Let us assume according to the import policy at router i, the routes for address prefixes AP_3, AP_4 and AP_9 as advertised by router x would not be accepted. Let neighbors y and z also advertise links, (AS_3, AP_3), (AS_4, AP_4) and (AS_5, AP_5) and the import policy accepts the links advertised by those two nodes. Therefore, the label set for link (AS_3, AP_3) would be $\{y, z\}$ while that of link (AS_5, AP_5) would be $\{x, y, z\}$.

The next step is to translate the metric policy (according to which higher preferences are given to some routes over others) to a quantitative form that facilitates the path selection process. Let us assume, according to the metric policy the route for address prefix AP_5 advertised by router x is given a lower preference, i.e. higher cost compared to the routes advertised for address prefix AP_5 by neighbors y and z. In order to translate the metric policy to a link-cost value, the label set of the incident link on address prefix AP_5 i.e. the label set of link (AS_5, AP_5) has to be modified and new links and nodes have to be added to the main topology, according to the method shown in Fig. 3.10. The route for AP_5 is given lower priority by adding redundant node R_5 and redundant links (AS_5, R_5) and (R_5, AP_5) (each with cost 5 and label-set $\{x\}$), while the label set of link (AS_5, AP_5) is changed from $\{x, y, z\}$ to $\{y, z\}$.

We have just described the first step for path computation, where the routes from neighbors are added to the topology database. Next step would be to build a

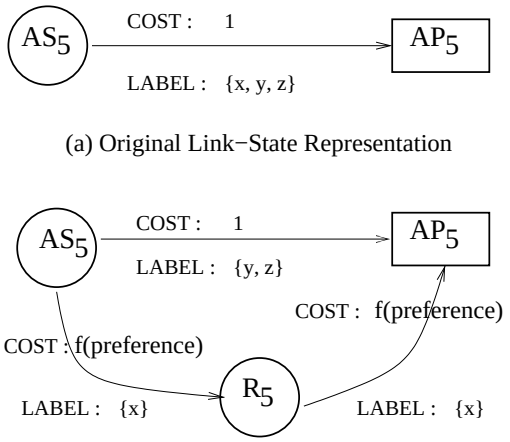


Figure 3.10: Method of conversion of metric policy to cost parameters

source tree by running the path-selection algorithm which has been described in the previous sections. After the source tree has been built, the export policy can be used to filter several parts of the source tree and the filtered source tree is then advertised to neighbors.

3.5 Conclusions

In this chapter we have described why a new path selection algorithm is necessary for computation of source trees when the network topology is built using on-demand link-state advertisements. We have shown that building a source tree that enables correct data packet forwarding is an NP-complete problem. Therefore, in SOAR or any on-demand link state routing protocol source graphs rather than source trees are required to be built at each node for computation of routes. For that purpose we have used a modified form of the Bellman-Ford algorithm.

We have also developed a polynomial-time approximation algorithm for build-

ing source tree when there is a constraint that for each destination only a subset of possible next-hops can be actually used. Subsequently we have used this algorithm for loop-free link-state policy based routing, where the basis of routing information exchange is a source tree and the forwarding structure maintained at each node is also a tree.

Chapter 4

On Demand Link-Vector Protocol (OLIVE)

In this chapter we are going to present the on-demand link vector (OLIVE) protocol. Like SOAR it is an on-demand link-state routing protocol. However, unlike SOAR it is guaranteed to be loop-free at every instant and does not depend on traversed path information to detect loops. After a change in network conditions, if a node running OLIVE finds that the selected route for a destination can lead to a loop, it forces its upstream nodes for the same destination to release it as successor. After the upstream nodes release it as successor, the node can choose a new path that is guaranteed to be loop-free. This process requires co-ordination only with one-hop neighbors, and unlike ROAM [36] does not require synchronization traversing multiple hops. OLIVE exchanges routing information in the form of paths in which each path consists of several links. The combination of paths advertised by a node to any neighbor form a

source graph in OLIVE and at any node the combination of paths advertised by its neighbors provides a partial topology of the network. This topology information is used at each node for selecting routes with active destinations, as well as for finding potential alternate paths when the original route breaks.

Section 4.1 provides the motivation and intuition behind OLIVE's design which originates in the implicit source routes attained in DSR [20]. According to implicit source routing, routers can attain loop-free routing by first exchanging path information in route requests and replies, and then performing packet forwarding over "default flows" for which no path information is needed in packet headers. Therefore, the question becomes whether one can improve on the exchange of path information among routers in such a way that a router can have as its default flow a multipath (multiple paths) that never contain a cycle. Sec. 4.2 provides a detailed description of OLIVE and illustrates its operation. Sec. 4.3 demonstrates the correctness of OLIVE. Sec. 4.4 compares the performance of OILVE with two on-demand routing protocols, DSR, AODV and two proactive routing protocols, TBRPF and OLSR. Sec. 4.5 gives our conclusions.

4.1 Motivation Behind Design of OLIVE

The main question that leads to the design of OLIVE is how should one design a loop-free routing algorithm that uses path information on-demand, although allows local route repairs and no source routes or flow identifiers in the headers of data packets?

OLSR and TBRPF are link-state based routing protocols that disseminate topology information about all routers and compute routes to each destination using a shortest-path algorithm. On the other hand, Spohn and Garcia-Luna-Aceves [24] have proposed STAR, a proactive link-state routing protocol in which each router receives "source tree" information from each of its neighbors consisting of links in the paths to all reachable destinations and forms a partial network topology and uses that for computing its own source tree.

We have already presented SOAR, which adapts the notion of source trees to on-demand routing. In SOAR, a router notifies a source tree to its neighbors that consists of paths to those destinations for which the router has traffic. It uses a modified form of Bellman-Ford algorithm to compute feasible routes to destinations, and creates a source graph at each router with the aggregate of the paths chosen to destinations. However, routers exchange source trees, and a router needs to force its neighbors to establish paths to destinations so that its source tree becomes valid. Therefore, in certain scenarios routes to destinations are exchanged in the form of trees, while the forwarding data structure is a graph. The key problem is that it is not free of temporary routing table loops. Consequently, data packets in SOAR carry path-traversed information to detect loops.

The question then is whether an on-demand routing protocol can be developed that is loop-free and uses source tree information.

In an on-demand routing protocol, a node may not know of any route for a destination, although based on physical connectivity, its neighbor might find that it is

on the shortest path to that destination. Therefore, to prevent incorrect packet forwarding the node cannot be used by its neighbor as the successor for that destination. This leads to several constraints in the choice of the shortest path for any path selection algorithm. These constraints have already been described in Chapter 3. Because of the constraints, while computing a source tree, finite cost paths for all destinations cannot be obtained, although some neighbors might have explicitly advertised those paths. In such a case, an optimal path-selection algorithm has to be developed that maximizes the number of nodes for which finite-cost paths can be obtained. This problem of finding the source tree with the maximum number of vertices has been termed as the OPT-TREE problem and has been defined in Chapter 3.

We have shown that the problem of computing a source tree using a routing protocol that exchanges path or link-state information on-demand, is NP-complete. Therefore, our design goal is to develop an on-demand protocol that exchanges path information without forcing the creation of a source tree. In other words, we must augment the use of path information implicitly adopted in DSR, which consists of a router choosing paths based solely on complete paths reported by its neighbors, rather than trying to adopt the approach used in proactive protocols based on link-state information, which merge the links of reported paths into a topology graph from which shortest paths are derived with a local path selection algorithm.

4.2 Description of OLIVE

4.2.1 An Overview

In OLIVE, a router sends route requests (RREQs) to establish routes for destinations with which it needs to communicate. The destinations, themselves or nodes having active routes to those destinations respond with route replies (RREPs). RREPs contain paths for destinations and are sent back towards the initiator of the route discovery process, very much like RREPs in DSR do. The aggregate of path information obtained in RREPs forms a partial topology.

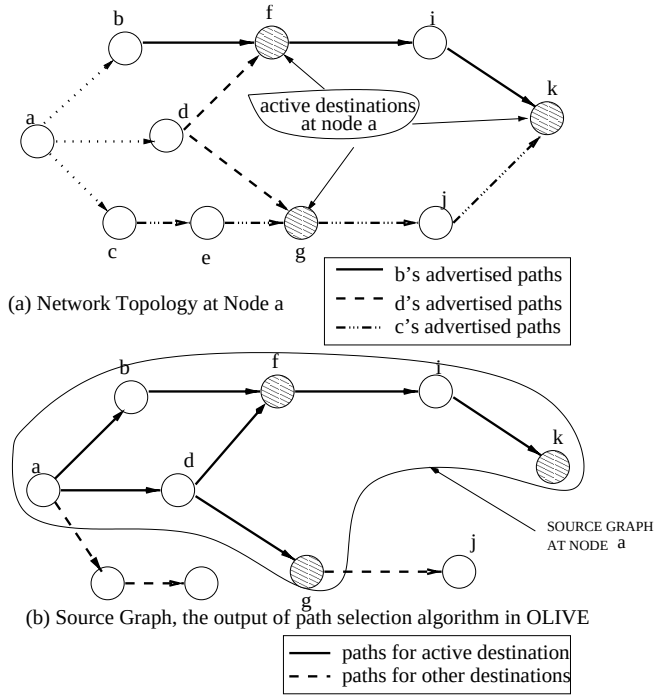


Figure 4.1: Path selection in OLIVE

A router selects paths for destinations for which it has active flows only from paths explicitly advertised by neighbors. The path-selection algorithm is: if $P_j^{n_i}$

($n_i \in N_i$, where Ni_i is the set of neighbors) is the path for destination j advertised to node i by neighbor n_i and $P_j^{n_i}$.cost refers be the cost of the path, then path $P_j^{n_x}$ is chosen as the active path if $P_j^{n_x}.\text{cost} = \min(P_j^{n_i}.\text{cost})$. Fig. 4.1 shows the method of computation based on the above path selection algorithm. Fig. 4.1(a) shows the network topology formed by combining paths advertised by neighbors b, c, and d. Node a has active flows with nodes g, f and k and therefore, needs to set up routes for them. For destination g, the advertised paths are $aceg$ and adg, of which the path adg is chosen because it is smaller in cost. Similarly among advertised paths $adfik$ and $acegjk$ for destination k, $abfik$ is chosen. For destination f, the only path possible is adf, because neighbor d has reported only this path for destination f.

For other non-active or active destinations for which there are no explicit path advertisements, the plausible routes are computed using the Dijkstra's algorithm, in which there is no constraint in the choice of the successor. In the example of Fig. 4.1 for non-active destination j, for which there is no path advertised explicitly by any neighbor, the plausible path $adgj$ has been selected using the Dijkstra's algorithm, which comprises of links advertised by different neighbors. Routes for destinations, for which no up-to-date explicit advertisement is present, are never installed in the main routing table. Forced routing is done first to check whether those paths are correct, and then data forwarding is carried out.

In Fig.4.1(b), the edges with solid lines form a source graph, where it contains routes for destinations g, f and k. When node a gets a route request for any of these destinations, it will reply with the relevant paths of the source graph. It forwards the

route request for other destinations.

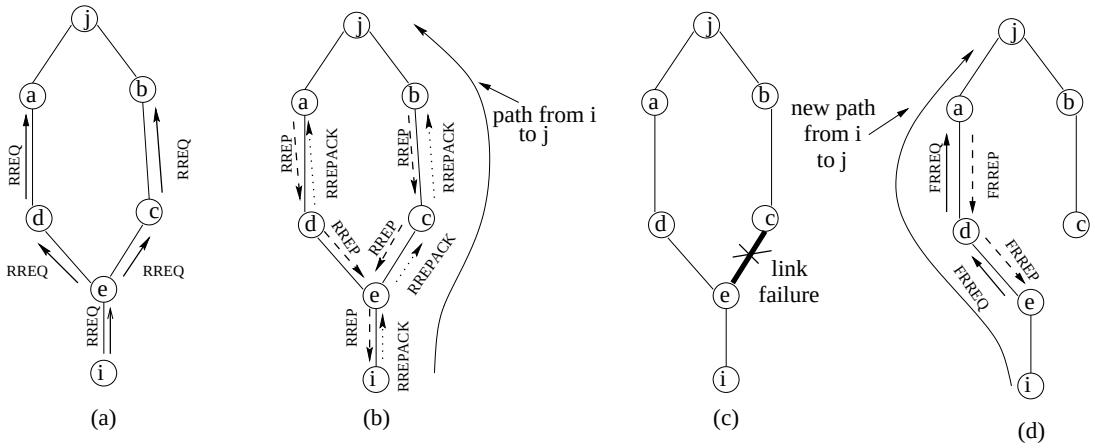


Figure 4.2: Route discovery and route repair methods in OLIVE

When a path breaks, an intermediate node attempts to locally repair the routes, i.e., establish an alternate path without informing its upstream neighbors. To do so, the node uses its partial topology information to find whether an alternate route exists. If such a path exists, nodes along the possible alternate path exchange *forced route requests* (FRREQs) and replies to FRREQs (FRREPs) to ensure the viability of that path. If the local route recovery process fails, route errors (RERRs) are sent upstream advertising link failures.

4.2.2 Example of OLIVE Operation

Fig. 4.2 shows an ad-hoc network of seven nodes. Each edge in the network connects those nodes who can hear each other directly. Assume that node i has data packets for node j and therefore, has to set up a route for destination j. Nodes a and b initially have active flows with node j and therefore, have valid routes for node j. Node i initiates a route discovery process for destination j. First it queries its neighbors for

a route and when it does not get a reply, it sends network-wide queries. Fig. 4.2(a) shows the flow of the Route Requests (RREQs) across the network, after it originates from node i. Fig. 4.2(b) shows how the route replies (RREPs), carrying routes for destinations and the acknowledgments to route replies (RREPACKs) are exchanged between routers for final path set-up between node i and node j. The RREQs contain information regarding the initiator and the target of route discovery process. The target or any intermediate node that has a valid path for the target can respond to the RREQ with a route reply (RREP). RREQs also carry information about the paths along which they travel and this information helps to route the RREPs back towards the initiator of route requests. In this example both nodes a and b have active routes for node j. Therefore, when each of these nodes receives a RREQ, it responds with a RREP containing a path for node j. Node a's RREP is meant for node d, while node b's RREP is sent to node c. When node a has sent a RREP to node d containing route to node j, it implies that it has a valid route for node j and it has included node d in the predecessor list such that node d can be notified when route for node j changes. If node a does not receive any acknowledgment from node d within a certain time interval, it removes node d from its list of predecessors. Therefore, when node d selects its route for node j through node a, it sends a RREPACK to node a. Similar messages are also exchanged between nodes c and b. At node e, RREPs containing paths to node j are received from both the nodes c and d. Both the paths are equal in cost; therefore, node e accepts the route it hears first. (When routes of unequal cost are present, shorter routes are always selected.) Let us assume, node e hears the

RREP from node c first. Therefore, node e sends RREPACK to node c only and not to node d. However, an entry for the advertised route of neighbor d will be present in node e's network topology and this information will be helpful to set up alternate paths, when the original route breaks. Node e sends RREP to node i which sends RREPACK back to node e after it selects the final path, $iecbj$.

Next assume that link (e, c) fails. Fig. 4.2(c) and Fig. 4.2(d) illustrate how alternate paths are created when original paths break due to link-failures.

Because successor c for destination j is no more reachable from node e, node e removes the route for node j from the routing table. After node e detects that its route to destination j has broken, first it attempts to locally repair routes based on its topology information without informing the upstream nodes. Node e finds from its network topology that there is a plausible path $edaj$ to reach node j. However, for destination j node e is not the current predecessor of node d. Therefore, node e may not have up-to-date path for destination j. Node e initiates the validation process, which we call forced routing in which nodes exchange messages to check the viability of the path. Forced Route Requests (FRREQ) get forwarded along the path, $edaj$ and any node (node a for this example) that has an active route sends a response (FRREP). FRREP is sent from node a to node d and then to node e and node e sets up the alternate route, $edaj$.

Unlike the case described above, if the situation would have been that local route recovery is not possible because of non-availability of any alternate route information, node e has to send a route error (RERR) to node i, containing failed route

information. Node i has to then restart the route discovery process, because it does not know of any alternate path.

4.2.3 Detailed Description

In this section we give the detailed description of OLIVE. The operations of OLIVE can be broadly classified into three phases: (a) route discovery for setting up new paths, (b) local route repair for finding alternate paths when the original breaks, and (c) route failure notification for updating neighbors of route failures.

Route Discovery

There are three types of control packets used during the route discovery phase: (1) RREQ, (2) RREP and (3) RREPACK.

Route Request (RREQ):

RREQ packets are used to discover routes for unknown destinations. Route discovery is always initiated by the source node, when it does not have routes for the destinations for which there are active flows.

Requests for routes by the source are either limited to neighbors or sent throughout the network. For the second case, each node forwards others' route requests and the route requests flow like an expanding ring search. Nodes forward a RREQ only if:

- The node has no valid route for the destination;
- No RREQ initiated by the same source has been forwarded within certain spec-

ified time;

- RREQ has not traversed beyond the zone within which the route search was limited.

Each RREQ contains information regarding the initiator and target of route discovery process and each forwarder of RREQ adds its own identity to RREQ's list of forwarding nodes such that this information can be used for forwarding replies back along the reverse path towards the source.

Route Reply (RREP):

Route replies are sent by nodes having active routes for the targets of RREQs and contain paths for the targets. RREPs are either sent in response to RREQs or forwarded in response to a RREP.

To avoid duplicate RREPs, a RREP sent in immediate response to a RREQ is broadcast to all neighbors. Before sending a broadcast RREP, a node waits for a back-off period (the length of back-off period is proportional to the node's distance from the target) and cancels the operation of sending RREP, if it receives RREP from another node, meant to be sent to the same source. When RREPs are forwarded, they are always sent unicast. Every node, before sending a RREP to a neighbor ensures that the latter is included in the predecessor list, so that it can be notified of route failures if the original route breaks.

When a route for a destination is installed in the routing table, it is given a lifetime of $active\_route\_timeout$, and a new predecessor entry has a lifetime of ($2\times active\_route\_timeout$). Every time a packet is forwarded for any destination, the

lifetime of the route to it is increased by *active_route_timeout*, while every time a predecessor forwards a data packet the lifetime of the predecessor's entry is increased by (2×*active_route_timeout*). The lifetime of the forward path to a destination is kept much smaller (half for our purpose) than the lifetime of a predecessor entry, so that upstream nodes are correctly updated about the changes in the forward path.

Route Reply Acknowledgment (RREPACK):

When a node establishes a route through a neighbor, using the path information just received from it, it acknowledges the neighbor with RREPACK. The neighbor on receiving the RREPACK cancels the ReplyAckTimer. After a RREP is sent ReplyAckTimer is started at the sender for each receiver of RREP and if no acknowledgment is received within a certain time frame, the timer times out and the predecessor entry for that receiver is removed.

Local Route Repair

By local route repairs, we mean when the original route breaks, intermediate nodes can repair the routes without informing the upstream nodes about the changes that are happening in the forwarding structure. This in effect reduces the number of mobile nodes that get affected due to network dynamics. When the source nodes, instead of flooding the entire network with RREQs, scope their searches among specific nodes, we call the process local route repair. Therefore, the local route repair refers to the process of repairing routes without always involving all nodes in the network.

Main two types of packets that are exchanged during this process are FRREQ

and FRREP.

Forced Route Request (FRREQ):

Alternate paths computed using network topology may not be up-to-date because they are not used recently for data delivery. Therefore, before the final data transfer FRREQs are sent along the alternate paths to check their viability.

Each FRREQ carries information about the plausible path to the destination such that each node on the path, having an active route can compare the path with its current route for the destination. The node either forwards FRREQ if it has no information about the viability of the path or responds with a FRREP if it is currently a source or relay for the destination and has definite information of the path. When FRREQs do not yield any response within a certain time interval, the alternate paths are assumed to be non-existent and information about failed routes is sent to predecessors.

Forced Route Reply (FRREP):

FRREP is sent as a reply to the FRREQ. It either contains a valid route for the destination or contains no route if the plausible path indicated in FRREQ is invalid. If the FRREP carries no path information, then information about the first link in the plausible path of FRREQ that is infinite is reported in FRREP. Therefore, the initiator of local route repair process will have a valid route or no route after receiving the FRREP and if no alternate route of equal or lower cost than the original is possible, RERRs (if it is relay) and RREQs (if it is source) are sent.

Route Failure Notification

When the original route for any destination breaks and no alternate path of equal or lower cost is possible, the router initiates the process of route failure notification. During this process the upstream nodes are informed of the changes in the routes, such that they either do the local route repairs or inform their predecessors. RERRs and RERRACKs are exchanged for route failure notification.

Route Error (RERR): Nodes send RERR to indicate route failures to their predecessors such that the predecessors are notified of the latest changes and they stop using them as successors. Failure of routes can happen under the following conditions:

- Adjacent Link Failure: The router can get notification of adjacent link failure from the network layer or link layer.
- Reception of Route Error (RERR): On reception of RERRs from neighbors, a node may find that its active route through those neighbors have broken.
- Expiration of Routes : When routes are not used for a certain time interval, they expire and the predecessors are notified of the expired routes using RERRs.

RERRs are also sent by a node to the sender of a RREP, when the RREP contains invalid routes.

RERRs for any destination contain information about the first downstream link in the original path that has become infinite.

Route Error Acknowledgment (RERRACK):

RERRACK is sent as acknowledgment to RERR to notify the sender of RERR that the node has stopped using the sender of RERR as successor. The sender of RERRACK is no longer considered a predecessor and when the number of predecessors becomes zero for a destination, the node will be able to install a new loop-free route, using the route discovery process.

4.2.4 Neighbor Relationship

Timely propagation of neighbor relationships is critical for efficient execution of a routing protocol. To obtain correct information, a router depends on notifications from the network layer or from the link layer.

At any node a link to a neighbor is up when any one of the following events happens:

1. the node receives directly a control packet from the neighbor for the first time
2. the node receives the first network layer hello message from the neighbor
3. A neighbor protocol at the link layer that monitors MAC layer traffic and also exchanges beacons, advertises the presence of the new neighbor.

A node decides that its link with a neighbor is down when any of the following events happens.

1. The router receives notification from the link-layer when it fails to deliver data packets along that link.
2. The network layer hello messages are missed several consecutive times.

3. A neighbor protocol at the link layer which monitors neighbor connectivity by sending beacons informs about a failed link.

4. Acknowledgments for data packets have not been received after repeated network-layer retransmissions.

If no network layer hello mechanism is available for neighbor discovery, and a neighbor is silent for a certain time (i.e., the router has not received any data packet or control packet from it for that period of time), the neighbor is assumed to be down and all link entries advertised by it are considered invalid.

4.2.5 Handling Link Sequence Numbers

Every node maintains its own sequence number, and when any of its adjacent links go up or down, it increments its sequence number and assigns that to the link, which has undergone the recent change. The cost of the link is infinite when the link is down, otherwise the cost is finite and can represent delay or capacity of the link.

Different neighbors might advertise different status for the same link. Under those conflicting situations, links are validated using sequence numbers and links with higher sequence numbers are trusted over links with lower sequence numbers. If there is no entry for a link, then the router trusts the first link-state entry it receives.

Links with infinite cost are never deleted and a node deletes information about a link with finite cost after each of its neighbors has removed that link from its advertised paths.

If a node deletes all information about a link with infinite cost and then re-

ceives link state information with finite cost but of lower sequence number than the sequence number of the recently deleted link with infinite cost, the result would be old link state information is injected for another time to the routers in the network. Because links with infinite cost should not be used for data delivery, extra control overhead is necessary to purge the nodes of old routing information. Therefore, bandwidth can be saved if failed link information is stored at each router.

4.3 Correctness and Loop Freedom

To prove OLIVE to be correct, following two conditions have to be satisfied:

1. **Safety Property :** Given a network $G = (V, E)$, and a destination $j \in V$, the successor graph $\mathbf{SG_j}(= (V, E^{'}))$, where $E^{'} = \{(i, s_i^j) : i \in V,\ s_i^j$: successor for node j at node i, **if** $s_i^j \neq NULL\}$, is a directed acyclic graph and hence loop free at every instant.

2. **Liveness Property :** Within a finite time following an arbitrary number of changes in network conditions and flows, all nodes in the network have correct paths for each reachable destination, to which they have active flows.

We note that the above conditions leave open the possibility of nodes trying to find persistently paths to destinations belonging to the partitioned network. In practice, the routing protocol can infer that a destination appears to be unreachable after a few failed attempts, and it is up to the higher-level protocol or application to determine whether or not to continue looking for paths to unreachable destinations.

The correctness proof assumes the following:

1. After an arbitrary sequence of link cost changes, topology changes, and traffic flow changes, there is a finite amount of time before the next change that is sufficiently long for OLIVE to stop computing routes to active destinations.

2. All information is stored correctly and routers operate according to the specifications of OLIVE.

3. Changes in status of adjacent links are notified within a finite time.

The instantaneous loop freedom in OLIVE can be proved based on the following two important properties:

1. When a node has to change its route, and the cost of the new route is higher than the cost of the old route, the node reports infinite distance to its predecessors and it changes to this new route only when the predecessors have removed it as successor. Change in successor following the selection of an alternate path of lower or equal cost do not lead to loops.

2. A node sets up a route for a destination only if the event that has forced the re-computation of route is reception of FRREP or RREP and the route chosen is the path specified in that RREP or FRREP. (As mentioned before, the sender of RREP or FRREP includes the recipient as the predecessor before sending the packets). This means that at any instant of time t at a node i, if $PRED_i^j(t)$ is the set of predecessors for destination j as known to node i, and $\overline{PRED}_i^j(t)$ is the set

of nodes at time t who have selected node i as successor to reach node j according to an omniscient observer, then $\overline{PRED}_i^j(t) \subseteq PRED_i^j(t)$. That implies if the original route breaks, a node if it is using the route will be definitely updated by its successor.

We can show by contradiction that loops can never form in OLIVE if the above two conditions are always satisfied. Let $a, b, c.....x$ be the nodes (Fig. 4.3(a)) which form a loop for a destination j with the next hop for node x being node a, that of node a being node b and so on. Let $P_i^j(t)$ be the path at node i for destination j at time t. Let us assume that at time t, a is the node in the loop whose length of the path for node j is maximum. Therefore, at time t, $P_a^j(t).\text{cost} \geq P_x^j(t).\text{cost}$. Assume that the last change in successor occurred at time $t' < t$, when node x chooses node a as the next hop. Therefore $P_x^j(t).\text{cost} = P_x^j(t').\text{cost} > P_a^j(t'').\text{cost}$ where $P_a^j(t'').\text{cost}$ is the cost of the path advertised by a in its RREP or FRREP at time $t'' < t' < t$. This implies $P_a^j(t).\text{cost} \geq P_x^j(t).\text{cost} = P_x^j(t').\text{cost} > P_a^j(t'').\text{cost}$, which basically says $P_a^j(t).\text{cost} > P_a^j(t'').\text{cost}$, i.e. node a has experienced an increase in the cost of path for node j in the interval $(t", t]$. In that case, according to the first rule given above, node a sends a route error to node x advertising infinite cost, in which case node x releases node a as next hop at time $t_x < t$. Therefore, the loop cannot form and that contradicts our assumption that there is a loop at time t.

Theorem 7 *Within a finite time after the last change in network conditions and traffic flows, all nodes which are sources of data packets have correct paths to the destinations.*

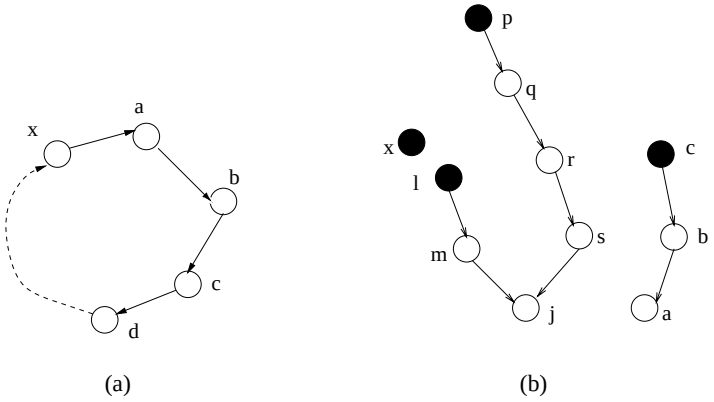


Figure 4.3: Loop-freedom and correctness of OLIVE

As described above, OLIVE ensures that the forwarding graph as visible to an omniscient observer, $\mathbf{SG_j}$ does not contain any transient loops. Let $S_j \subset V$ be the set of nodes that have active flows with destination j, and $R_j \subseteq V$ be the set of nodes which act as relays for the data packets to node j. Let t_1 be the time when SG_j is correct and loop-free. Then there are link cost changes and there are introduction and termination of traffic flows. Let t_2 be the time when the last traffic flow change or the last link change occurs. Here we want to prove that within a finite time after t_2, the paths from every $v \in S_j$ converge to j. In this context, we define the active graph $\mathbf{AG_j}$ to be the graph formed by the links (v, s_j^v) for each $v \in S_j$ or $v \in R_j$ (s_i^j : successor for node j at node i). It is obvious that $\mathbf{AG_j} \subseteq \mathbf{SG_j}$. So $\mathbf{AG_j}$ is guaranteed to be a DAG but for correct operation of OLIVE, we show that $\mathbf{AG_j}$ is a tree (rooted at node j).

Assume that this theorem is not true. Therefore, $\mathbf{AG_j}$ is not a tree and can be a forest, as shown in Fig. 4.3(b). In Fig. 4.3(b) the black nodes of $\mathbf{AG_j}$ denote the members of set S_j while the other nodes shown act as relays. Because $\mathbf{AG_j}$ is not a

tree, the path from any node $v \in S_j$ either (1) converges to j, or (2) does not have a path for j, or (3) ends in a node with no path for j.

We have to consider the last two cases, as they contradict the theorem. To illustrate, according to Fig. 4.3, the active graph for destination j shows that (a) after finite time node a does not have a path for node j and (b) nodes c and b think they have a correct path for node j even though the downstream node a does not have a path.

If node b is using node a as successor to reach j, it implies that node a has reported a path of finite distance for node j to node b and included node b as a predecessor for the route to node j. Node j is an active destination for node a and node a has no path for node j. Then, node a have reported path loss to node b. Therefore, within a finite time, node $b$receives the message that node a does not have a path for node j and because the path to any destination j through a neighbor n has to be reported by node n, it implies that node b will stop using node a as next hop within a finite time. So it is easy to show by induction that node c ($\in S_j$) will have no path to node a within a finite time; and all intermediate nodes will know about path losses to j. Therefore, only the following scenario is possible that can disprove the theorem: any node v ($\in S_j$) either has no path for j.

Within a finite time after t_2, a node $v \in S_j$ does not have a route for j, which implies that v generates infinite number of RREQs. The assumption here is that the network is not partitioned and the network consists of finite number of nodes, there exists at least one node that generates infinite number of RREPs and there exists

at least one node (that can include the source v) that receives an infinite number of RREPs, but does not get a new path. It implies the path sent in each RREP is invalid at the receiving node and that can only happen if the advertised path contains invalid links i.e., the receiving node has links with infinite cost but of higher sequence number. According to the rule of OLIVE, if after receiving a RREP a node does not get a path to node j, the receiving node sends to the sender a RERR advertising the invalid link. So within a finite time the sender gets corrected and will have latest link state information. Because the network consists of finite number of nodes, within a finite time all nodes, which have outdated link state information and advertised that in RREPs, will get updated. The link information with infinite cost does not get aged out or deleted. Therefore, the process of correcting neighbors cannot continue for infinite time. Within a finite time at least one RREP will be accepted at each node v ($v \in S_j$) and each node v will have the correct path for destination j.

4.4 Simulation Results

We have compared the performance of OLIVE with two on-demand routing protocols (DSR [25], AODV [34]) and two proactive routing protocols (TBRPF [30], OLSR [22]), which have all been proposed for standardization in the IETF working group on mobile ad-hoc networks (MANET) [17]. The performance evaluation has been done in the ns2 simulation platform [19], using the code of DSR, AODV and TBRPF provided with the simulator. TBRPF code conforms to version 4 of the Internet draft. For OLSR, we have used the code available from INRIA website [4]

Table 4.1: Constants used for OLIVE

Constants	Value
active_route_timeout	5.0 s
predecessor_lifetime	2 × active_route_timeout
olive_rreq_gap_time	0.4 s
olive_rrep_gap_time	0.4 s
rerr_ack_wait	0.5 s
reply_ack_wait	1 s
one_hop_traversal	0.1 s
frreq_time_out_value	2×one_hop_traversal × tot_hops

and have added the code for handling link-layer notifications of adjacent link-failures. The specifications of OLSR code match those in version 3 of OLSR Internet draft. The constants for DSR, AODV and TBRPF have been used unchanged from the original code, while the constants given in Table 4.1 have been used for OLIVE. The AODV code conforms to the specifications mentioned in the version 3 of the Internet draft of AODV. DSR code conforms to the version 1 of the Internet draft of DSR. The link layer implements the IEEE802.11 distributed co-ordination function (DCF) for wireless LANs. The broadcast packets are sent unreliably and are prone to collisions. The physical layer approximates the 2 Mbps DSSS radio interface (Lucent WaveLan Direct-Sequence Spread-Spectrum [46]). The radio range is 250m and for all the simulations the run length is 600 seconds.

TBRPF, DSR, AODV, OLSR and OLIVE use link layer indications about the failure of links when data packets cannot be delivered along a particular link. Except for the notification of the link layer about links going down, none of the protocols has any other interaction with the lower layer. In particular, promiscuous listening was disabled for both DSR and OLIVE. ARP has also been disabled and for the sake of

simplicity, the IP addresses of the nodes are used as the MAC addresses.

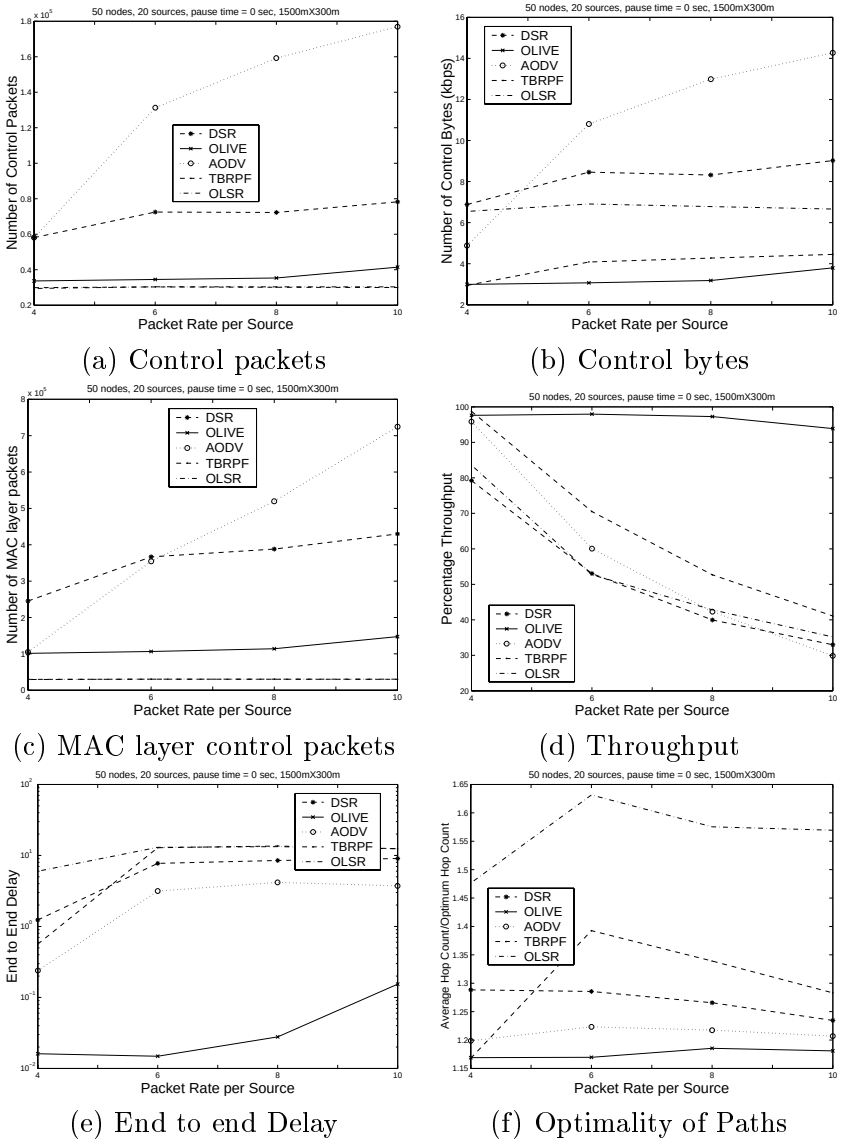


Figure 4.4: Performance in a 50-node network with 0 second pause time and 20 sources with varying packet load

4.4.1 Mobility Pattern

The movement of the nodes in the simulation is according to the random waypoint model [9]. We have described previously in Chapter 2 the details of random waypoint model. For our simulations we have 50 nodes moving over a rectangular area of 1500m×300m.

Values of *pause time* used are 0, 15, 30, 60, 120 and 300 seconds.

4.4.2 Input Traffic Pattern

Each flow is a peer-to-peer constant bit rate (CBR) flow and the data packet size is kept constant at 512 bytes. Each flow continues for 200 seconds and after the termination of the flow, within 1 second, the source randomly chooses another destination and starts another flow, which again lasts for 200 seconds.

4.4.3 Comparison Criterion

The following metrics are used to compare the performance of the routing protocols:

- *Packet delivery ratio*: The ratio between the number of packets sent out by the sender application and the number of packets correctly received by the target destinations. The main objective of on-demand routing protocol is to achieve maximum throughput using minimum control overhead.

- *Control packet overhead*: The total number of control packets sent out during the simulation. Each broadcast packet is counted as a single packet. Low control

packet overhead is desirable in low-bandwidth wireless environments.

- *Control byte overhead*: The total number of control bytes used in the control packets. This gives an idea regarding the total bandwidth used by control packets.

- *Total number of MAC packets* : The total number of packets sent at the link-layer for exchange of routing information. They include RTS, CTS, the control packets and the ACKs. This parameter can give an idea regarding the amount of battery power that is used for delivery of control packets. If the MAC layer protocol is TDMA, then the total number of control packets would be equal to the total number of packets sent by MAC layer. In such a scenario the control byte overhead will reflect the total energy usage for setting up routes.

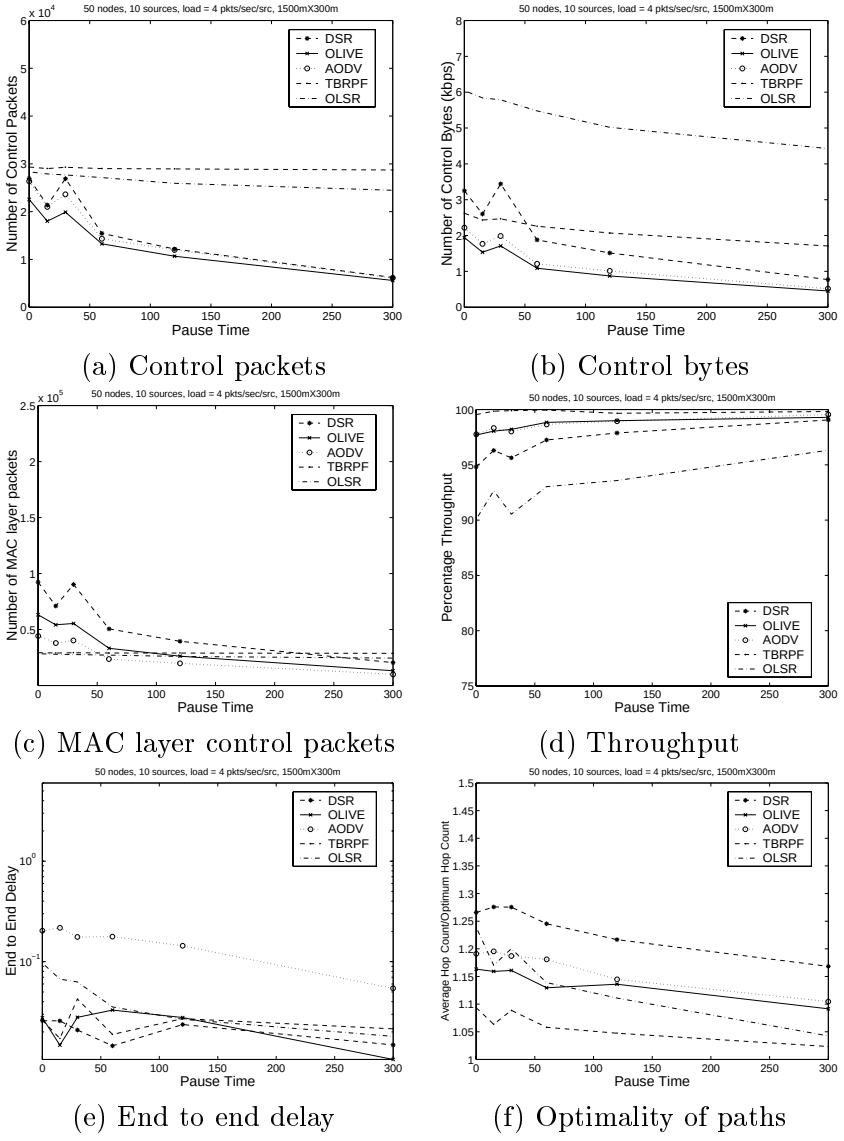
- *Optimality of paths*: Ratio of the actual number of hops to the optimal number of hops possible based on the given topology.

- *Average end-to-end delay*: The end-to-end delay measures the delay a packet suffers after leaving the sender and then arriving at the receiver application. This includes delays due to route discovery, queueing at IP and MAC layers and propagation in the medium.

4.4.4 Effect of Increasing Load

We have evaluated the performance of each of the routing protocols under varying packet load when the number of sources is 20 and nodes are constantly moving

(Fig. 4.4).



(a) Control packets (b) Control bytes

(c) MAC layer control packets (d) Throughput

(e) End to end delay (f) Optimality of paths

Figure 4.5: Performance in a 50-node network with load of 4 packets/s/source and 10 sources under varying mobility

The control overhead of OLIVE remains unchanged with the increase of load. DSR, TBRPF and OLSR exhibit similar behavior. However, AODV's control overhead continuously increases with load. Even when nodes are physically close, due

to congestion the links are assumed to fail when data packets can not be delivered along those links. In such cases increased number of route failures affects AODV most because for each route failure it generates network-wide queries. (AODV increases sequence numbers when there is a route failure and therefore, nodes with valid routes and lower sequence numbers can not respond. That leads to increased dissemination of route requests). The number of MAC layer control packets increases slightly for DSR throughout all scenarios while for OLIVE it increases only when the network load is maximum (41 kbps/source or 10 packets/source). This is because of the increased number of broadcast RTSs, for which there is no collision avoidance mechanism. Hello packets form the major percentage of packets for TBRPF or OLSR, therefore its control overhead stays unchanged. In terms of control overhead, there is no significant difference in performance between TBRPF or OLSR.

In terms of application-oriented metrics like throughput or network delay (Fig. 4.4(d) and Fig. 4.4(e)), the performance of OLIVE is the best among all protocols and remains almost unaffected by network load, while the performances of DSR and AODV degrade significantly. OLIVE, unlike DSR, TBRPF or OLSR, when it detects a route is broken, uses FRREQs and FRREPs to find the viability of alternate paths and data packets are forwarded only if the paths are usable. On the other hand, for TBRPF, DSR or OLSR, packets are rescheduled along different paths, without testing their feasibility leading to higher waiting time in queues and more congestion. OLIVE has a high range of delay values, like DSR, TBRPF, OLSR or AODV, which implies that data packets in OLIVE also have long waiting time at the link-layer interface.

However, its 95 percentile delay value is far lower than that of DSR, OLSR, AODV or TBRPF, which shows that on an average it has better delay performance. Network topology information in OLIVE also helps it to find shorter paths (Fig. 4.4(f)).

For TBRPF, data packets always get re-scheduled along alternate routes, when the original paths break. Therefore, they get circulated throughout the network and there is huge amount of packet loss due to looping and TTL timeout. In TBRPF, all the control packets are broadcast. Therefore, for sending any TBRPF control packet, no extra MAC layer handshake is necessary. However, because the broadcast TBRPF packets can lead to collisions, its performance degrades significantly with load. Same is the case for OLSR, whose proactive mechanism of route maintenance is almost similar to that of TBRPF. This is also true for AODV, where queries which form majority of its control packets are broadcast packets.

4.4.5 Effect of Mobility

Fig. 4.5 and Fig. 4.6 show the performance of all the protocols for a 50-node network with 10 sources and 20 sources respectively with each source generating packets at the rate of 4 packets/s.

Except for TBRPF and OLSR, the control overhead of OLIVE, DSR and AODV decreases with lesser mobility of the nodes. Higher mobility implies higher rate of route failure leading to higher control overhead. For TBRPF and OLSR, the control overhead remains almost unchanged with mobility, because the periodic hello packets which are always sent irrespective of the reliability of links form the majority

of the control packets. For 10 and 20 sources, the control overhead of OLIVE in terms of both bytes and packets is less than DSR, AODV, TBRPF or OLSR.

As we have seen before, because the majority of control packets in AODV is broadcast queries that does need RTS/CTS to send data packets, the difference between AODV and OLIVE in terms of MAC layer control packets (Fig. 4.5(c) and Fig. 4.6(c)) is not so significant as the difference in the number of network layer control packets.

In general when the number of sources is ten, the on-demand routing protocols use fewer network layer control packets compared to proactive routing protocols, while for the high mobility scenarios with higher number of sources proactive routing protocols start performing better. In all scenarios, OLIVE has less control overhead than DSR or AODV.

OLIVE delivers the same number of data packets as TBRPF or AODV (Fig. 4.5(d) and Fig. 4.6(d)). DSR suffers a higher loss of data packets in all scenarios, because of the use of stale routing information in the RREPs. OLSR suffers considerable loss of data packets due to routing loops and TTL timeouts.

In terms of path optimality (Fig. 4.5(f) and Fig. 4.6(f)), OLIVE is best among all the on-demand routing protocols. However, because TBRPF uses hello packets, it can learn about a node which has moved closer faster than the on-demand protocols that depend on reception of control packets to detect neighbor connectivity. In all our experiments we have found that the paths in DSR are always longer than the paths in AODV, contrary to the results in [9, 10]. The reason is that because of no

promiscuous listening DSR cannot learn about shorter routes

In terms of delay (Fig. 4.5(e) and Fig. 4.6(e)), OLIVE performs better or equal to the other protocols. Queueing at link layer is the main cause for the delay experienced by data packets in each of the routing protocols.

4.4.6 Looping Problem

Here we quantify the amount of bandwidth wasted in each routing protocol due to packets going in loops or staying in the network for a considerable time. The experiment is done for low to heavy load scenarios when the number of sources is 20. Saving bandwidth is essential in ad-hoc networks especially when the load is heavy, because then there is increased contention in the medium and more queueing at each layer.

Fig. 4.7 shows the number of packets that have actually gone in a loop, or have been dropped due to TTL timeout and have been dropped on detection of loops. Loops can be detected in two ways in any routing protocol, in which no traversed path information is present in the data packets: (a) the source finds that the data packet has come back to it (b) any forwarding node detects that it is passing the packet to a node, which has actually forwarded the data packet.

In our experiments, the packet traces are used to detect the number of packets that have gone in loops. From Fig. 4.7, we see that in both TBRPF and OLSR, bandwidth is wasted due to looping and the effect becomes more prominent with heavier load. Both OLSR and TBRPF are proactive link-state protocols, and their

routing exchange mechanisms do not ensure instantaneous loop freedom. Also they always exchange unreliable control packets and that leads to longer convergence time. Therefore, a significant amount of bandwidth is wasted by packets going in loops. Both these routing protocols have minimum packets dropped due to non-availability of routes, which implies that the topology information always helps the data packets to be re-scheduled along alternate paths. However, in heavy load scenarios, when the links fail frequently due to congestion, alternate paths are not always the correct options. The control packet exchanges in OLIVE and AODV ensure instantaneous loop freedom. In DSR, the source routes carry information about path traversed and the path to be traversed. Therefore, in DSR loops cannot form. Under high load, in DSR some data packets go into loops when data packets are salvaged at intermediate nodes. When an intermediate node in DSR finds that the next link in the source route is no longer available, it salvages the data packets by re-routing the packet using its own cached routing information. Because path traversal information is not checked for re-routing, loops can form.

Though AODV and OLIVE ensure instantaneous loop freedom, looping of packets can still occur during the transient states, when routes change. However, due to no routing loops, this effect is not persistent for AODV or OLIVE.

4.5 Conclusions

We have presented the on-demand link-vector (OLIVE) protocol, which is the first protocol to ensure loop-freedom at every instant using the same path infor-

mation available in DSR while allowing destination-based incremental routing instead of requiring source routing of data packets.

We have shown that selecting paths on-demand cannot be approached based on the "source trees" used in proactive routing protocols. Therefore, routers in OLIVE exchange path information and these paths combine to give the partial network topology. A path selection algorithm is then run on the network topology to compute the source graph. Source graphs are reported incrementally in the form of separate paths. Topology information provides routing data to limit network-wide searches for routes and do local route repairs. OLIVE has been shown to be loop-free at every instant and to find correct paths to destinations in finite time.

Our simulation results show that OLIVE performs much better than two popular on-demand routing protocols DSR and AODV, and two proactive routing protocols like TBRPF and OLSR in terms of control overhead, throughput or delay.

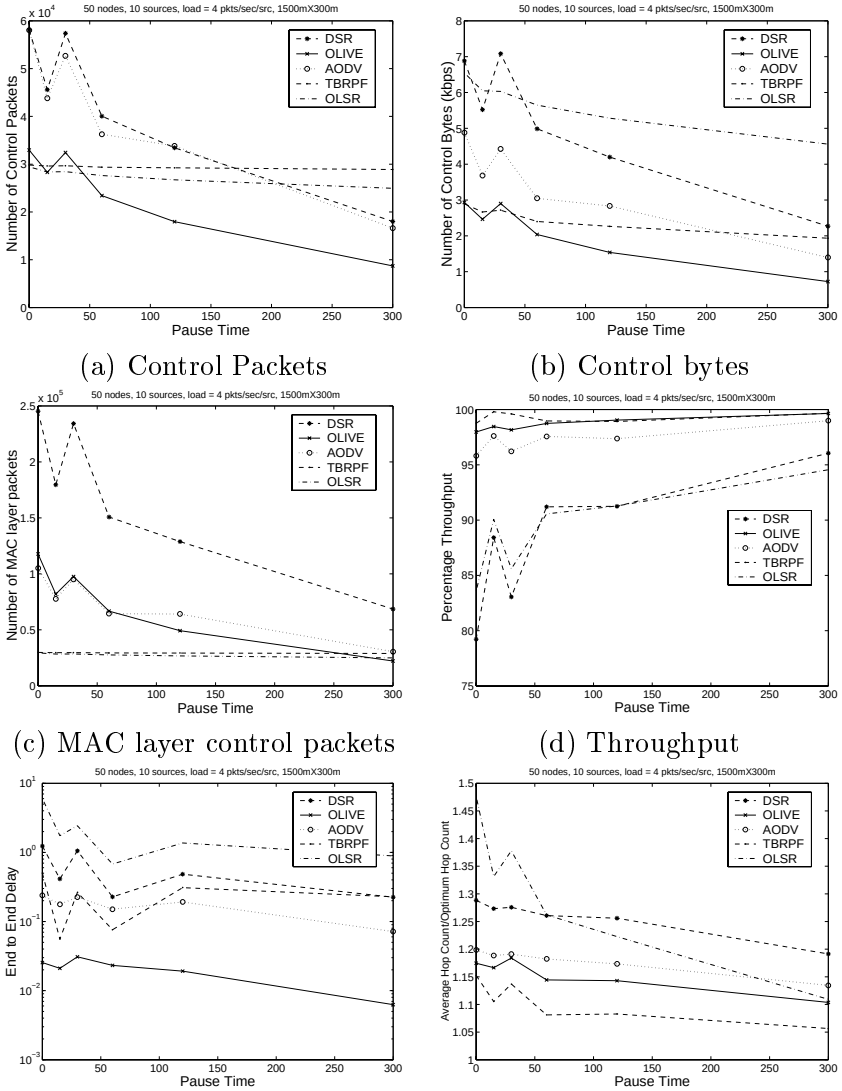
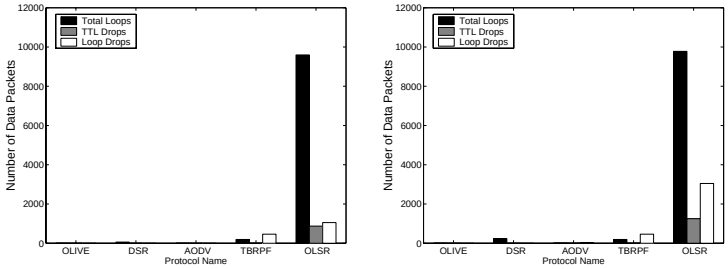


Figure 4.6: Performance in a 50-node network with load of 4 packets/s/source and 20 sources under varying mobility



(a) Rate = 4 packets/s (b) Rate = 6 packets/s

(c) Rate = 8 packets/s (d) Rate = 10 packets/s

Figure 4.7: Loops for a 50-node network with 20 sources under four different load conditions

Chapter 5

Node-Centric Hybrid Routing

We have presented routing protocols for ad-hoc networks that use information about destinations of traffic flows to set up on-demand routes. However, none of these protocols or other state-of-the-art routing protocols designed for ad-hoc networks do not make any assumption regarding the distribution of data traffic in the network. In this chapter, we will make a realistic assumption regarding the type of data traffic that can exist in the practical scenarios of ad-hoc networks and then propose a new genre of routing that can enhance the performance over pure on-demand routing protocols.

Table-driven or proactive routing protocols can become expensive in mobile ad-hoc networks. Control overhead in the proactive routing protocols increases with the size of the network and becomes redundant if the number of communicating peers is many fewer than the total number of nodes in the network. To address the scaling problem of table-driven routing, on-demand routing protocols have been proposed for ad-hoc networks. Nodes running such protocols set up and maintain routes to destina-

tions only if they are active recipients of data packets. When routes are not available, network-wide queries are generated to establish routes to destinations. However, when only a few nodes of the ad-hoc network must act as sources and sinks of data packets, maintaining routing information to such nodes on-demand and treating those nodes as any other node may not be as attractive as a proactive approach to establishing routing information to them while on-demand routing is used between less accessed nodes. This motivates the interest in a node-centric hybrid approach to routing in ad-hoc networks, where traffic would be mainly from common nodes towards certain nodes that provide special services like Internet connectivity to all other nodes.

The Zone Routing Protocol (ZRP) [15] constitutes a framework for hybrid routing in ad-hoc networks. ZRP adapts a hierarchical-routing approach based on clusters (called zones) and maintains routes proactively to destinations inside a zone, and on-demand routing is used to establish routing information spanning more than one zone. We advocate a different approach to hybrid routing that is node centric rather than based on zones of the network.

We call those nodes that support special services for the rest of the nodes (and therefore have a high likelihood of communicating with the rest of the ad-hoc network) *netmarks*. This scenario is illustrated using Fig. 5.1, where there is a single netmark and an ad-hoc network of mobile nodes $a, b, ..., t$. The netmark can be fixed as well as mobile depending on the type of ad-hoc networks. Under a node-centric hybrid routing approach, paths (shown with solid lines) will be constantly maintained between $a, b, c, ..., t$ and the netmark. If node e wants to communicate with node f and

node j talks with node p, the paths between those nodes are set up in an on-demand basis and are shown with dashed lines in the figure. Observe that node c does not need to know how to reach node r, so no route for node r is created at node c. The most important advantage of this approach is that it is an enhancement over the basic on-demand routing protocol such that it can utilize the non-uniform distribution of traffic, if it is present, to improve performance. However, the basic on-demand routing protocol can be used unchanged when the traffic distribution is random.

The landmark hierarchy [45] is an earlier node-centric approach to hierarchical routing designed for proactive routing in large networks. The key difference between node-centric routing described and the landmark hierarchy is that a landmark becomes the address of a common node, while a netmark is a destination that provides services.

Section 5.1 introduces two approaches to node-centric hybrid routing. Section 5.2 describes how SOAR can be modified to adopt the principle of hybrid routing. Section 5.3 describes the challenges in the routing approaches when there are multiple netmarks in the system. Section 5.4 shows through the simulation results the advantages of node-centric hybrid routing over pure on-demand routing protocols in the ad hoc networks, which act as wireless extensions of the Internet. Section 5.5 concludes the chapter.

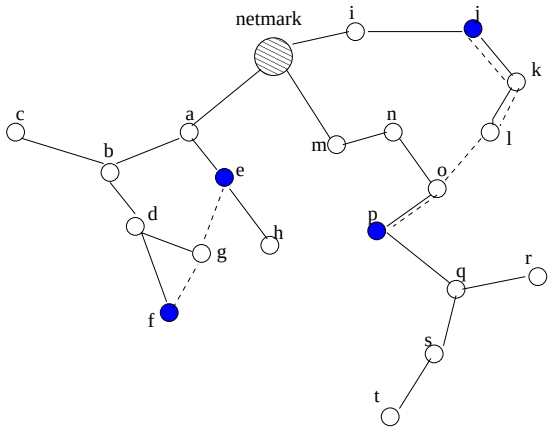


Figure 5.1: An ad-hoc network with a single netmark

5.1 Approaches to Node Centric Hybrid Routing

5.1.1 Extended Caching of Netmarks

In pure on-demand routing protocols, routers set up paths to other nodes based on the existence of flows with them. Routes are cached once they are obtained using a route discovery mechanism and they are modified when they become invalid due to link failures. Among the on-demand routing protocols proposed in literature, the basic difference is in how routes are cached and invalidated, and how route changes are reported to other nodes. Here we present an extension to the rules of on-demand routing protocols as an approach towards node-centric hybrid routing, in which in the absence of data traffic the routes for netmarks are stored and maintained for longer periods of time compared to the caching time used for common nodes. Let that time interval be *cache_time*. This leads to the following two main changes to the basic mechanism in which pure on-demand routing protocols work:

- during *cache_time*, it sends a route request for a netmark whenever the router

looses all routes to the netmark, irrespective of whether there exists any flow with the netmarks.

- During *cache_time*, route errors are generated for netmarks by intermediate nodes when they loose their routes for them, irrespective of whether there exists any flow with the netmarks.

Here we focus on how we can adapt the idea of extended caching in existing on-demand routing protocols, namely AODV, DSR and SOAR.

The ad-hoc on-demand distance vector (AODV) protocol uses sequence numbers to prevent permanent loops and relies on hop-by-hop routing, rather than source routes. RREQs are generated by the sources of data packets and forwarded by intermediate nodes. When RREQs get forwarded, reverse routes for the source are installed. Route Replies (RREPs) can be sent by the destination or an intermediate node with an unexpired route entry for the destination. The RREP message initiates the creation of a path for the destination in intermediate nodes that forward the RREP back to the sender of the RREQ. Each routing table entry has an expiration period (*active_route_timeout*) associated with it.

The specification of AODV [34] states that AODV can use the periodic network-layer hello packets or link-layer notifications for determining connectivity with neighbors. When a node detects that its path to a destination is broken due to failure of a link with a neighbor, it sends a route error (RERR) packet to its active predecessors for that destination.

AODV can easily incorporate the idea of extended caching by increasing the expiration period of the route for the netmarks to a higher value than that assigned to the expiration period for the routes for other common nodes in the network. Therefore, every time the paths to a netmark fail due to link failures, the nodes can send RERRs to active predecessors, which travel upstream. When a node looses route for a netmark, it starts the route discovery mechanism irrespective of the presence of traffic flows for netmarks. Royer et al. [43] suggest a pro-active foreign agent discovery mechanism using AODV, but do not modify the basic route repair technique of AODV for the foreign agents, such that the routes to foreign agents remain current even in the absence of data traffic.

The dynamic source routing (DSR) protocol uses source routes to forward data packets and exchanges routes in the form of paths [21]. Routes are stored in a cache, until an indication that a link in the route is broken is obtained through route error (RERR) messages or link layer notifications. A route discovery cycle is started by a source if it looses all routes to the required destination. DSR can determine on its own whether a link is broken by doing multiple retransmissions, or can depend on the link layer for link failure notifications.

DSR keeps routes for destinations until RERRs or link layer notifications indicating failure of routes are received. These events are triggered only by the failure of transmission of data packets over links, which implies that route failure propagation will not improve by extended caching of netmarks, in which the basic idea is to maintain correct paths in absence of data traffic. However as in AODV, DSR can be

easily modified to send RREQs for netmarks, irrespective of the presence of traffic.

As described in Chapter 2, the source tree on-demand adaptive routing (SOAR) protocol is a link-state routing protocol in which routers exchange minimal source trees in their control packets, consisting of the state of the links along the paths used by the routers to reach active (important) destinations. Important destinations are determined at any node based on whether it is currently carrying any data packet for it. The basic principle by which on-demand routing works in SOAR can be easily extended to incorporate the idea of extended caching of netmarks. The only change needed for SOAR is that netmarks would be considered important for longer periods of time than the common nodes and depending on the level of importance of a particular node, paths to nodes will remain fresh for different intervals of time. We call this modification netmark-aware on-demand link state routing (NOLR).

5.1.2 Proactive Routes to Netmarks

Here we present the second approach to hybrid routing in which proactive routes are maintained for the netmarks while on-demand routes are used with other nodes. The modifications required for any on-demand routing protocol to adopt this approach are the following:

1. Adding a route for a netmark for the first time necessitates sending updates to neighbors, so that they can also set up new paths to the netmarks.

2. Route errors and route requests are generated for netmarks irrespective of the presence of traffic to the netmarks.

3. A netmark can advertise its presence by sending hello packets to enable new neighbors to set up paths to it, and to help the old neighbors check whether the netmark is still reachable without having to depend on link-layer notifications i.e. the presence of data traffic. This makes paths to netmarks proactive, rather than being data-packet driven.

Now we look at how DSR, AODV or SOAR can be changed to incorporate the above concepts of hybrid routing. Implementing a network-layer Hello mechanism at the netmarks is easy in any of the three protocols. hello packets sent by the netmarks in AODV should contain the highest sequence number for the netmarks, so that the receiving node can install new routes for the netmarks.

Propagation of new route information for netmarks in SOAR only requires a change in the rules for sending an update. SOAR, in its pure on-demand routing form send updates when there is an increase in distance in any of its routes. Here the change in the rule that is necessary is that the node should send updates when it discovers routes for the first time. SOAR does not need a new type of control packet while both AODV and DSR need the introduction of a new broadcast control packet that propagates paths to netmarks when a route is first found for the netmarks. As described earlier, SOAR contains routes to important nodes. Therefore source trees in SOAR always contain links to netmarks, irrespective of the presence of traffic.

Because DSR sends RERRs only to the source of data packets, adding a hello mechanism at the netmark will not help DSR, because it is not clear how failure information for netmark-routes can be propagated to other nodes in the network when

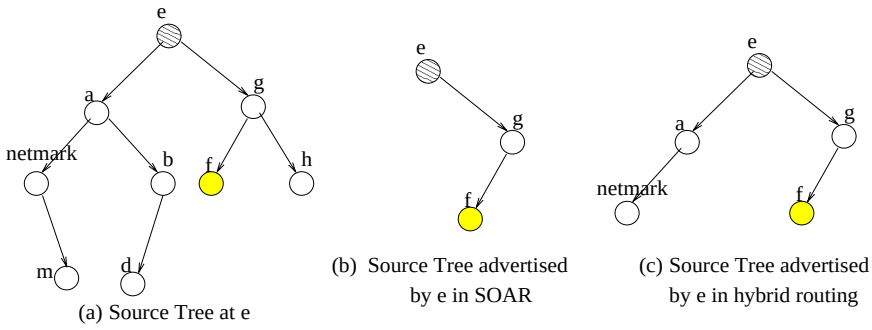


Figure 5.2: Difference in control information in SOAR and node-centric hybrid routing protocols like NEST or NOLR

the route failure is determined by loss of hello packets, rather than by the failure of delivery of data packets. An improvement can be achieved if a node keeps track of the neighbors who use it for data delivery to netmarks for the last *pre-defined* amount of time, so that route failures to netmarks can be reported to those predecessors.

5.2 Netmark-aware Source Tree Routing (NEST)

To illustrate the benefits of node-centric hybrid routing over pure on-demand routing SOAR is chosen to incorporate the above two approaches for node-centric hybrid routing. The reasons behind this :

1. As shown in Chapter 2, SOAR is more efficient than DSR and AODV.

2. Modifications required in SOAR to adopt hybrid routing are much simpler than in both DSR or AODV. As discussed above, DSR may not be able to take advantage of the proactive or extended route maintenance for netmarks because its route maintenance mechanism is not purely data-packet driven.

As discussed earlier, extended caching of netmarks can be adopted in SOAR by considering paths to different destinations as important for different periods of time. We call the modified version of SOAR as netmark-aware on-demand link-state routing (NOLR).

The netmark enhanced source tree (NEST) routing protocol incorporates in the basic routing mechanisms of SOAR the changes needed to maintain proactive routes for netmarks and on-demand routes for other nodes in the network. Next we provide the details of NEST.

Fig. 5.2 shows the difference in the control message between NEST and SOAR for the network shown in Fig. 5.1. In NEST, unlike in SOAR every node has a proactive path with the netmark and common nodes e and i have on-demand routes set up for nodes f and p respectively. The source tree at node e (Fig. 5.2(a)) is the tree consisting of links that node e uses to reach the netmark and other nodes in the netmark. Router e advertises a part of this complete source tree to its neighbors, which is called the *minimal* source tree. For SOAR the *minimal* source tree would only consist of links needed to reach nodes with which it has active flows. In this example, because node e has active flow with node f, the minimal source tree advertised by node e would be as shown in Fig. 5.2(b). In NEST, even if node e does not have active communication with the netmark, it advertises links belonging to the path to it (as shown in Fig. 5.2(c)), along with the links of the path to node f.

In case of bi-directional flows, the netmarks need to establish reverse routes to all the common nodes. Because the netmarks need to communicate to each mobile

node in the network, finding routes to such nodes using a route discovery mechanism is not an attractive approach. In NEST, the facts that (a) nodes maintain loop free proactive routes to the netmark, and (b) data packets toward a netmark are forwarded along that route, can be utilized to set up the reverse paths. The solution is based on the assumption that the links are bi-directional in nature. When an intermediate node forwards a data packet from any source towards a netmark, it can record in its routing table the previous hop from which it has received the data packet as the next hop towards the source. This entry is kept soft state and is removed if the route is not refreshed by another data packet within a predefined amount of time. This process enables the netmark and the intermediate nodes to know the next hop to reach any mobile node, without incurring extra control overhead. In on-demand routing protocols, the path towards a source of query is set up according to the route traversed by the query. The reverse path in NEST is similarly validated by the forward path traversed by the data packets.

5.2.1 Netmark Discovery

In NEST, netmarks send hello packets to inform their neighbors of their presence. Netmark advertisement can be done by sending beacons at the MAC layer. When a node receives a hello packet, it assumes the presence of netmark in its neighborhood or the neighbor protocol can send such indication. At the routing layer, if the hello packet is lost several consecutive times, then the router can declare that the link to its neighbor is down. The MAC layer can also notify the routing layer about

link failures when it cannot deliver data packets. The later mechanism helps to detect link failures faster than using the hello mechanism.

When a node has a new entry for the netmark, it updates it neighbors about the new route, which in turn will choose to send an update if it discovers the netmark for the first time. Therefore, through recursive advertisement every node in the network will know about the path to the netmark. For example, in Fig. 5.1, when node a learns about the existence of the netmark in its neighborhood, it advertises its path to it, and its neighbors b and e re-advertise. The updates in NEST are broadcast packets and hence unreliable. If the updates are lost, some nodes may not know about the route to the netmark. When a node comes up, it will also not be aware of the routes to netmarks. Under such circumstances, NEST will initiate queries. Because paths to netmarks are maintained proactively by all nodes, complete flooding might not be necessary always.

Another important issue to consider is how to make the nodes in the network aware of the identities of netmarks. This can be done in two different ways :

- The netmarks are known previously and all common nodes are pre-configured statically with the identities of netmarks.
- If the nodes that have netmark status are not fixed, then nodes that obtain netmark status can advertise this fact by sending hello packets and all the nodes can mark the netmarks in the control packets. When a node ceases to be a netmark, it can notify all nodes by flooding the information that it is no longer a netmark. This situation can happen in relief scenarios or battlefields, where

the group leaderships change over time.

In this context another issue is to decide which nodes can be netmarks. For the ad-hoc networks which are wireless extensions of the Internet, this case is simple. The gateways or DNS resolvers can be netmarks. Otherwise, depending on the applications different conditions can be checked for deciding which nodes can be netmarks.

5.2.2 Maintaining Bi-directional Paths

In the previous section, we have seen how every node in the network establishes a route for the netmark. When data packets are forwarded along the path from any source to a netmark, reverse routes can be set up as soft state from the forwarder towards the source node. Fig. 5.3 illustrates this process.

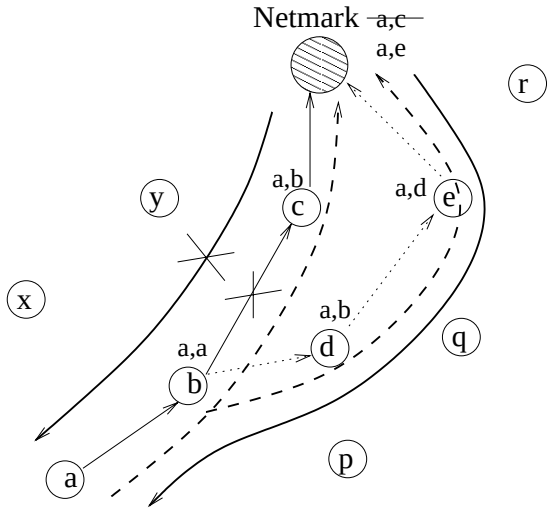


Figure 5.3: Setting up of paths between netmarks and nodes

In Fig. 5.3, when node c learns of the netmark, it advertises the netmark

in its source tree and therefore, node b will know about the netmark and also about neighbor c. When node b re-advertises, node a will know about the path to netmark and the intermediate nodes b, c. Similarly, node b will know about an alternate path $[b, d, e, netmark]$ from neighbor d, but node b chooses the path through node c because it is of smaller length. If link (b, c) fails, then node b can choose the alternate path to netmark through neighbor d. When node b advertises its source tree, it may not advertise the link to node a because node a is not an important destination. In such a case, node c will know only about node b, but not about node a. Similarly, the netmark may know about node c, but not about nodes a and b. In order to send data packets to node a, the netmark in such a case would have to initiate a route discovery for node a. To prevent a query from being initiated by the netmark for every common node in the network, the following mechanism is adopted to set up the reverse paths without introducing any extra control overhead.

When data packets start flowing from a node toward the netmark, the intermediate nodes along the path toward the netmark can set up paths toward the source of the data packets. For example, when the data packet from node a reaches node c from node b and finds that the destination is a netmark, then node c adds an entry in its routing table for node a as $[destination = a, nexthop = b]$. Similarly, the netmark will keep an entry $[destination = a, nexthop = c]$. These routing entries will expire after a *soft_state_interval*. When link (b, c) breaks, data packets will be forwarded along the path $[a, b, d, e, netmark]$ and netmark will replace the route for node a, $[a, c]$ with $[a, e]$ when the data packets arrive from node e. Similarly, when

nodes d and e forward packets, they set up soft-state entries for the destination a. Node c removes entry $[a, b]$ after the *soft_state_interval* due to the absence of any data packets from node a towards the netmark from node b. A node changes the next hop for the reverse path towards any destination only if the previous route has not been used for $soft\_state\_interval$. This prevents route-flapping in case routes for different netmarks from the same source pass through the same set of intermediate nodes. This situation can arise due to changes in network conditions and delayed reporting of route changes.

The reverse routes are set up towards the source based on flow of data packets only if the destination of the data packets is a netmark. This is because each node maintains up-to-date paths only to netmarks, and because the paths from a node to any other node may not be current, a similar reverse path set up process would lead to stale routes. Assuming links are bi-directional, during the steady state, the reverse route from a netmark is essentially the same as the forward path. Therefore, the reverse paths are going to be correct because the forward routes to the netmarks are correct. In the absence of data packets, when there is no forward flow, the netmark must resort to queries to find paths to the destinations. However, in case of bi-directional flows when data starts flowing in both directions the route needs to be established only for the first time using queries. So if the flow stays continuously ideally no further control packets should be needed to maintain the reverse paths. This mechanism of reverse path set up and maintenance also ensures symmetry of the paths taken by data packets, which is beneficial for bi-directional flows like *tcp* flows.

5.2.3 Packet Forwarding

When a packet is received from the application layer and it is meant for a node in the ad-hoc network, it forwards the packet provided that it has a route. When the packets are meant for a node outside the subnet of ad-hoc network, the common nodes would determine that by looking at the IP address of the destination. And in such a case the data packets would be encapsulated and the new destination would be the netmark. The netmark would decapsulate the packet and forward it to the destination, which can be in the wired Internet or another ad-hoc network.

5.3 Multiple Netmark Scenarios

When multiple netmarks are present in the network, depending on the purpose the netmarks serve, the routing can be adapted to further improve performance of routing protocols. How to do this depends on the way in which nodes affiliate themselves to netmarks.

5.3.1 Static Affiliation

Irrespective of the position of a node with respect to the netmark, a node can be made to always use a particular netmark. This can happen in a battlefield scenario, where a group of mobile soldiers always communicate with the group leader. In those scenarios, irrespective of the position of a soldier, each soldier has to report to his or her group leader. Depending on the mobility of a node, in order to reach the netmark affiliated to it, it might have to use another node, which is affiliated to a

different netmark, which implies every router has to know the routes to all netmarks, though packets from a node would always be forwarded to a particular netmark.

5.3.2 Dynamic Affiliation

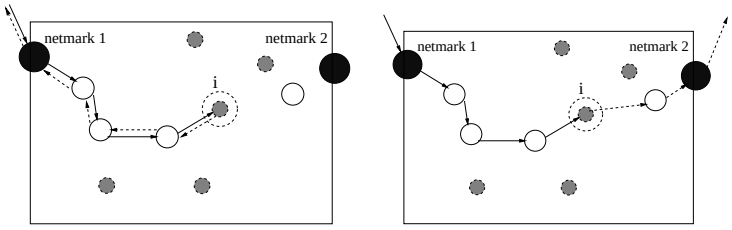
In this scenario, a node need not be affiliated with any particular netmark. This can happen when the ad-hoc network is an extension of the Internet, and there are multiple Internet access points and a common node in the ad-hoc network can communicate with any access point to reach the wired Internet. Because packets can be forwarded to any netmark, routing becomes efficient in terms of control overhead because (a) redundancy of routes to the Internet helps to reduce the number of route requests for the netmarks; (b) paths taken by outgoing packets within the ad-hoc network become shorter; and (c) anycast route discovery mechanism can help to reduce the control overhead. Queries are not required to be sent individually for each netmark. *Anycast* queries can be sent asking for a route to the anycast address of all the netmarks. In such a case, any router who has a route to any netmark can reply and in case of availability of multiple routes, the reply would contain the route to the nearest netmark. This process will reduce number of queries, path lengths, and time to discover routes.

5.3.3 Hybrid Affiliation

There can be scenarios where the affiliations of the nodes can be both static and dynamic. Packets for some nodes are always forwarded to certain fixed netmarks,

while packets for others can be forwarded to the nearest netmark. For a example, it is easier to communicate with a netmark that hosts a proxy server in its subnet rather than those netmarks which do not have the proxy server in their networks. Packets meant for networks remote to any of the netmarks' networks can be forwarded to any of the access points. The routing information maintained in the case of hybrid affiliation would be the same as in static affiliations.

Depending on how the packets enter the ad-hoc network and how the outgoing packets are forwarded, paths followed by data packets can be symmetric or asymmetric. In case of static affiliation, as shown in Fig. 5.4(a), if the incoming packets for a common router enter the network through the netmark with which it is always affiliated, then the data-path can be symmetric. For the example shown in Fig. 5.4(b), due to dynamic affiliation though the data packets arrive for router i from *netmark 1*, the outgoing packets are forwarded towards *netmark 2*, thereby making the data-path assymetric.



(a) Static Affiliation (b) Dynamic Affiliation

Figure 5.4: Data paths in an ad-hoc network with multiple netmarks

When the netmark is an Internet access point, it will advertise routes to subnet but not host routes. So it may happen $netmark1$ advertises path to subnet, which includes the identity of node i. However, due to network partitioning, $netmark1$

may not have any route for node i. Therefore, packets for node i are dropped if they are forwarded to $netmark1$. However, node i can still be reached through $netmark2$. Therefore, if the netmarks form a fully connected overlay network using the wired Internet, reachability can be improved even with partitioning in the wireless part of the network.

5.4 Performance Evaluation

We evaluate using simulations whether node-centric hybrid routing approaches can give improved performance over pure on-demand routing protocols. We have compared the performance of node-centric hybrid routing approaches, NEST and NOLR with pure on-demand routing protocols SOAR, DSR and AODV using the ns2 network simulator (ns-2.1b6 [18]). For AODV, we have used the code available from Marina et. al. [27]. The AODV code conforms to the specifications mentioned in the version 3 of the Internet draft of AODV. However, the values of constants used for AODV are according to the values given in the code. DSR code conforms to the version 1 of the Internet draft. SOAR has been implemented according to the specifications described in Chapter 2. NOLR is the modification of SOAR that does extended caching of routing information for netmarks. NEST uses the same values for the constants common with SOAR. The values of NEST-specific constants are shown in Table 5.1.

In Table 5.1, Hello_Interval is the interval between the sending of consecutive hello packets by the netmark while Dead_Time_Interval is the time interval during which if a netmark remains silent, the adjacent nodes assume that the link to the

Table 5.1: Constants for NEST

Hello_Interval	:	3 s
Dead_Time_Interval	:	9 s
soft_state_duration	:	1 s

netmark is no more valid. In case a broadcast control packet is sent, the netmark defers the next transmission of hello packet for Hello_Interval seconds. The constant soft_state_duration is the maximum time a soft-state routing entry can stay in the routing table without being refreshed. As explained before, soft state entries are maintained in response to data packets that flow from a common node to the netmark.

DSR, AODV, SOAR, NOLR and NEST do not depend on link layer for neighbor discovery. All protocols use link-layer indications for link failures when data packets cannot be delivered along a particular link. Use of a neighbor protocol at the link-layer for discovering neighbors can significantly improve the performance of routing-layer protocols. However, because our objective is to test the routing protocols as stand-alone protocols, we have not considered the effects of MAC layer interactions on the routing protocols' performance. Promiscuous mode of operation has been disabled. As described in Chapter 2, promiscuous listening improves the performance of any routing protocol that does not depend on a neighbor protocol, but practical implementation of promiscuous listening faces several obstacles. The link layer protocol used is the IEEE802.11 distributed co-ordination function (DCF) for wireless LANs, which uses RTS/CTS/DATA/ACK pattern for all unicast packets. Broadcast packets are always sent unreliably. The physical layer approximates the 2 Mbps DSSS radio interface (Lucent WaveLan Direct-Sequence Spread-Spectrum [46]). The range

of the radio is 250m. In our experiments at the start of the simulations all the nodes know the identities of the netmarks and the netmarks do not loose their special status anytime during the entire length of the simulations.

5.4.1 Mobility Patterns

Nodal movement in the simulation occurs according to the random waypoint model introduced by Broch et. al. [9] and described in Chapter 2.

5.4.2 Traffic Patterns

We have introduced two different traffic models for performance evaluation, which we call the INTNET model and the RELIEF model. These traffic models are more realistic compared to the traffic models used in [9, 8], where continuous CBR traffic flows exist between randomly chosen nodes making the traffic pattern more or less uniform throughout the network. Moreover, the data packets in a flow do not follow a continuous CBR pattern, the flows in INTNET and RELIEF model assume that packets can come in bursts and in between bursts there can be zero data packets.

The INTNET model is applicable for ad-hoc networks that are wireless extensions of the Internet. The communication is mainly from each of the common nodes towards the netmark which can host commonly-accessed servers or act as the access point to the Internet. In comparison to the number of flows between nodes and netmark, the flows strictly between common nodes is much smaller in number. The traffic pattern between any common node and the netmark is based on a FLOW_OFF/ON

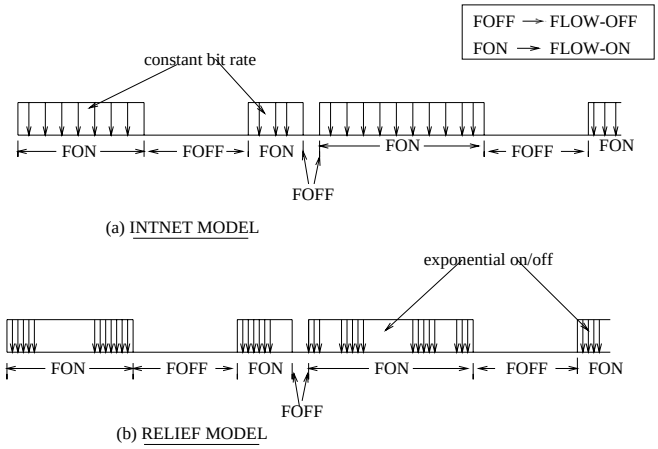


Figure 5.5: Traffic flow scenarios

Table 5.2: Specifications for INTNET model

FLOW_ON period	:	Uniform Dist (30,120) s
FLOW_OFF period	:	Uniform Dist (50, 120) s
Packet Size	:	66 bytes
Rate	:	2, 3, 4 ,5 packets/s per node

model, as shown in Fig. 5.5(a). The parameters of the INTNET model are shown in Table 5.2.

During the FLOW_ON period, there exists a *cbr* traffic and there is no packet flow during the FLOW_OFF period. The motivation behind simulating the INTNET model against a model in which the flows are ON continuously is that web traffic [7] consists of FLOW_ON/OFF periods where the OFF periods correspond to the user's think time, while the ON period represents download time. In our experiment with the INTNET model at all times there are four random flows between two randomly selected nodes. The duration of these flows is always 200 seconds. All the flows are bi-directional.

Table 5.3: Specifications for RELIEF model

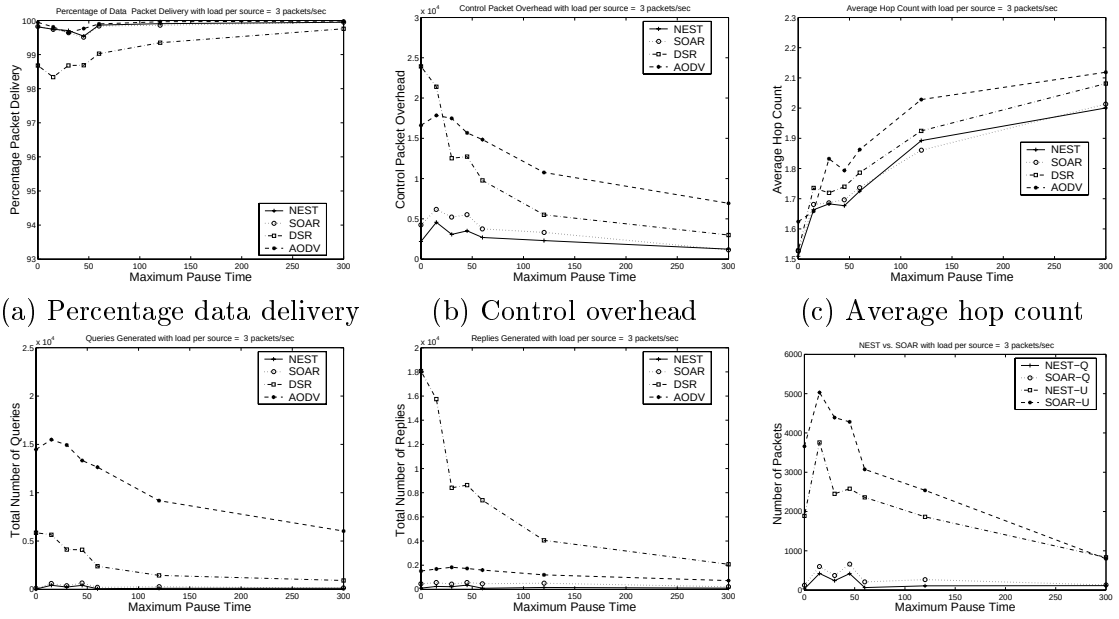
FLOW_ON period	:	uniform (30,150) s
FLOW_OFF period	:	uniform (10, 20) s
Packet Size	:	66 bytes
Rate [2]	:	17 packets/s (9 kbps)
talkspurt [2]	:	350 ms
silence [2]	:	650 ms

RELIEF traffic model has been introduced to simulate traffic in relief or battlefield scenarios where the group members report to the group leaders while the group members also exchange information. The group leader is the netmark who is contacted more frequently compared to other nodes. There are four random flows only between common nodes and at most six random flows from a common node towards the netmark. We have logically divided the set of common nodes into five groups and only one member in the group can talk at a time with the netmark. The packet arrivals during the FLOW_ON period follow an interrupted deterministic process (IDP) as shown in Fig. 5.5. The IDP model has been used to simulate the voice traffic. ON/OFF periods during a FLOW_ON period correspond to talkspurt/silence period of the speaker. The parameters for the RELIEF model have been specified in Table 5.3.

5.4.3 Performance Criteria

We have evaluated the routing protocols based on the following metrics: (a) Packet delivery percentage, (b) control packet overhead (c) average hop count (d) end-to-end delay (e) overhead due to queries (f) overhead due to replies.

5.4.4 Experimental Scenario 1

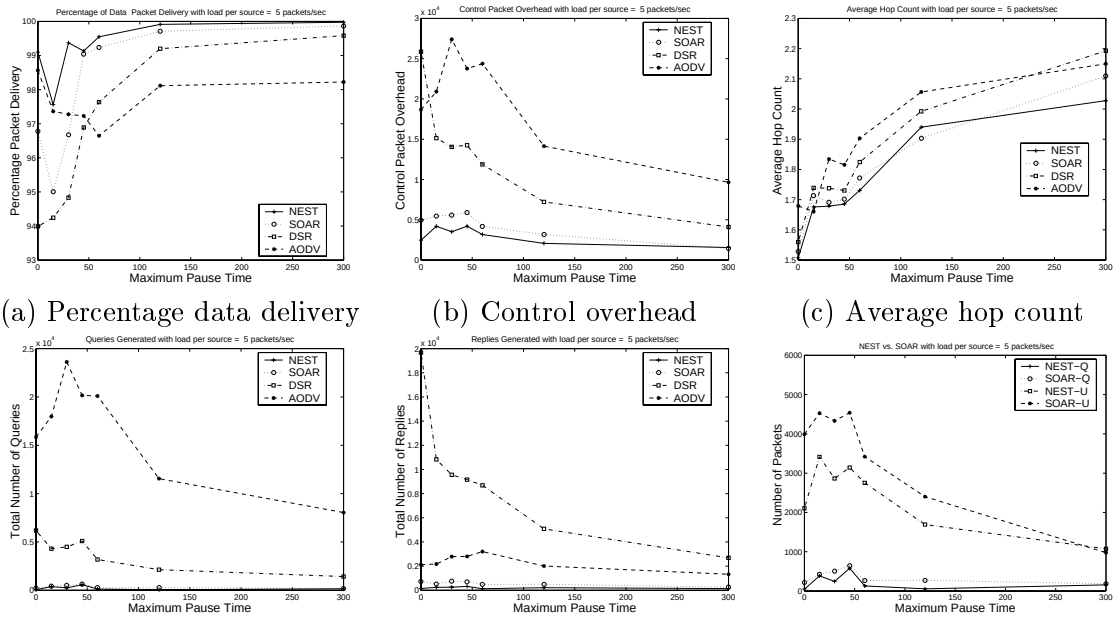


(a) Percentage data delivery (b) Control overhead (c) Average hop count

(d) Total queries (e) Total replies (f) NEST vs. SOAR

Figure 5.6: Performance of NEST, SOAR, DSR, AODV in a 31-node network with a fixed netmark at load generated per node of 3 packets/s

This scenario consists of a network of 31 nodes moving over a rectangular area of 1000m×500m. There is a single netmark in the system, which is placed at co-ordinates (500, 250) and is fixed throughout the length of simulation. The *pause time* during one simulation run is uniformly distributed between zero and a maximum value. The possible maximum values for *pause time* are 0, 15, 30, 45, 60, 120 and 300 seconds. The simulation length is 600 seconds, while the results are presented on the basis of at least 3 simulation runs where each run is having a different randomly generated mobility scenario but same traffic model (this is also true for subsequent experiments). The traffic model is according to the INTNET model. The performance results have been presented for two different load scenarios of three and five packets/s



(a) Percentage data delivery (b) Control overhead (c) Average hop count

(d) Total queries (e) Total replies (f) NEST vs. SOAR

Figure 5.7: Performance of NEST, SOAR, DSR, AODV in a 31-node network with a fixed netmark at load generated per node of 5 packets/s

per node during the FLOW_ON period.

Most of the findings on AODV, SOAR and DSR from our experiments conform to the results published previously [8, 9] and to those results described in Chapter 2. As shown in Fig. 5.6(b) and 5.7(b), AODV's control overhead is significantly higher than DSR's or SOAR's control overhead except for the high mobility scenarios. AODV's control overhead consists primarily of queries (Fig. 5.6(d) and Fig. 5.7(d)), while the control overhead of DSR consists mainly of replies (Fig. 5.6(e) and Fig. 5.7(e)). This is because AODV resorts to route discovery more often than DSR, while DSR sends multiple replies to queries. Contrary to previous findings [8, 9] we have observed that AODV's control overhead in highly mobile scenarios is lower than DSR's. Because each node in the INTNET model sends and forwards packets for

a netmark, each node in DSR ends up having more cached entries for the netmarks as opposed to the case where the netmark is accessed by only a small number of sources. That effectively leads to a significantly high overhead of cached replies and when the cached entries become stale, the effect of cached replies becomes counter productive. In low mobility scenarios where path information becomes stale less often, the number of replies is reduced due to fewer route requests and due to the reduction of the effect of injection of old routes due to multiple replies.

SOAR sends fewer control packets compared to DSR or AODV under all mobility scenarios with varying loads. This is because SOAR resorts to less route discovery because it utilizes redundancy in control information and during link-breakages minimizes the number of affected nodes by locally repairing routes. Because SOAR and DSR both can use stale information, under heavy load scenarios and high mobility, SOAR and DSR suffer slightly more in terms of data delivery compared to AODV. The effect is less in SOAR compared to DSR because SOAR uses sequence numbers to validate link state information while in DSR explicit route error messages are required to invalidate link-state information.

NOLR performs same as SOAR. Therefore, though we have also tested the performance of NOLR in our simulation experiments, for clarity we have not included any results of NOLR in Fig. 5.6 and Fig. 5.7. Unlike SOAR, NOLR maintains routing information for netmarks for longer periods of time compared to the time for maintaining information for other nodes. In our simulations because each node all the time either sends or forwards packets for netmark, any node in SOAR also ends up treating

the netmark *important* throughout the simulation. So the advantage due to prolonged caching is not easily noticeable.

Under all scenarios, NEST has been found to perform much better compared to any other purely on-demand routing protocols, both in terms of data delivery and control overhead. NEST (Fig. 5.6(a) and Fig. 5.7(a)) always delivers more data packets compared to other protocols, with the effect being more prominent under heavy load. In NEST each node always maintains proactive paths to the netmark. Therefore, the paths to netmarks are always more up-to-date in NEST than in a pure on-demand routing protocol. SOAR maintains information for netmarks for significant length of time; however, NEST paths are more accurate, because the netmark advertises itself periodically to force its routing information in other nodes and nodes using NEST update their neighbors when they first discover routes to netmarks. This conclusion is validated by the results of Fig. 5.6(f) and Fig. 5.7(f), where we see that more updates (NEST-U and SOAR-U) are needed in SOAR compared to NEST to purge wrong link state information. On an average, NEST produces around 30% fewer updates than SOAR. We also find that NEST uses fewer number of queries (NEST-Q and SOAR-Q in Fig. 5.6(f) and Fig. 5.7(f)) compared to SOAR, which is desirable in large networks where queries can be expensive. The reduction of queries in effect causes reduction of replies in NEST (Fig. 5.6(e) and Fig. 5.7(e)). Queries are sent by NEST for common nodes always and for netmarks during network partitioning. We also see from Fig. 5.6(c) and Fig. 5.7(c) that on an average, hop count in NEST is the smallest. This is because, the hello mechanism helps to detect faster the direct links to netmarks

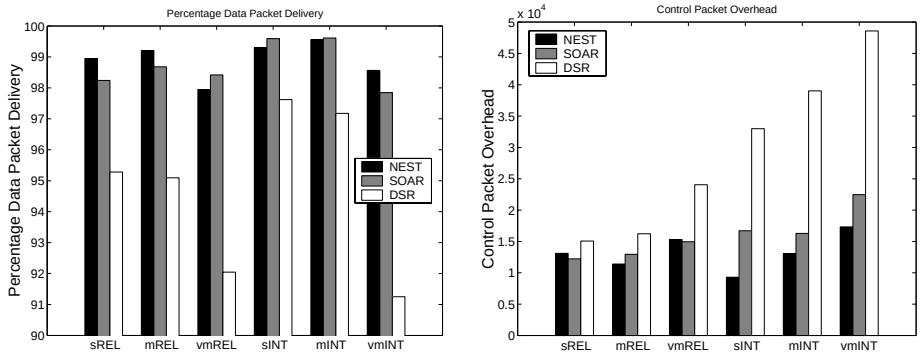
and hence the paths can get shortened.

5.4.5 Experimental Scenario 2

This scenario consists of a network of 30 nodes and one netmark with common nodes moving with speed selected from a uniform distribution between 5m/s and 20m/s. (*Pause* time of the netmark is uniformly distributed between 0 seconds and 30 seconds). Three different movement scenarios for the netmark are analyzed while keeping the mobility pattern for other nodes the same. The netmark in these scenarios is either static (model s), mobile (model m : the netmark moves over a rectangular area 250m×250m) or very mobile (model vm : the netmark can move over the entire area). Motivation behind using different mobility models is that in relief scenarios the group leaders, which are the netmarks can be either static or mobile. Because most of the traffic is towards the netmark, the routing protocols would be more stressed to maintain routes when the mobility of the netmark increases. The netmark moves with speed similar to the speed of other common nodes with pause time between 10 seconds and 30 seconds. We use two different traffic models : INTNET and RELIEF, which are respectively indicated as INT and REL in Fig. 5.8. A static netmark model with INTNET traffic pattern is indicated as $sINT$, while a vm netmark model with RELIEF traffic pattern is represented as $vmREL$.

From Fig. 5.8 we see that there is no appreciable difference in performance between the m and s model. This is because the radio range is 250m and the netmark moves over an area of (250m×250m) for the m model, which does not contribute to

too many additional path changes. The performance of each of the routing protocols suffers when the netmark becomes very mobile. From Fig. 5.8, we find that INTNET model produces more stress on the routing protocols than the RELIEF model with DSR being affected the most. Though the traffic pattern is different in both cases, the total number of data packets sent throughout the simulation is the same and the network does not get overloaded.



(a) Percentage data delivery (b) Control overhead

Figure 5.8: Performance in a 31-node network with three mobility models for netmark and two traffic models

From Fig. 5.8(a), we see that, in terms of data delivery, SOAR and NEST deliver on an average same number of data packets in both traffic models, which we have also seen in our results of Sec. 5.4.5 for the low-load scenarios. DSR's percentage data delivery is 4%-7% less than SOAR or NEST, with the performance becoming worse with higher mobility of the netmarks. This is mainly because DSR has stale path information and suffers from packet losses when data packets get forwarded to nodes who have actually deleted those routes.

In terms of control overhead, DSR's control overhead is comparable to that of SOAR or NEST for *sREL* or *mREL* models (Fig. 5.8(b)). However, DSR sends

significantly more control packets for the INTNET model because in INTNET model the number of flows is more.

SOAR and NEST have similar control overhead for the RELIEF model, although in the INTNET model, NEST outperforms SOAR and DSR. This is because the RELIEF model has fewer flows (about six) toward the netmark compared to INTNET model. NEST depends on the flow of data packets for detecting link failures for routes between common nodes. Therefore, when the number of flows in the network decreases, stale links due to less usage of links for data packet delivery make the routing information for NEST outdated and force NEST to send same number of updates as in SOAR. This indicates that the node-centric approach to proactive route maintenance for special nodes can improve performance with higher differentials with increased amount of communication from mobile node towards the netmark.

We also see that the proactive route maintenance in NEST does not suffer in higher degree than DSR or SOAR when netmarks are very mobile, which could have been an argument for using purely on-demand approaches in these practical scenarios.

We have analyzed the delay performance for the voice traffic in each experimental scenario (Table 5.4). The results presented are for a randomly chosen run, so as not to average out the high frequency components of individual runs. This is important for voice traffic performance because worst-case performance results limits the quality. The conclusions that can be drawn from the data of Table 5.4 are:

- Range is significantly high under all cases. This is because the time between two route discovery cycles increases after each unsuccessful attempt and when

Table 5.4: End to end delay distribution of voice traffic for different netmark mobility models

	Static Relief Model (sREL)			Mobile Relief Model (mREL)			Very Mobile Relief Model (vmREL)		
Percentile	NEST (s)	SOAR (s)	DSR (s)	NEST (s)	SOAR (s)	DSR (s)	NEST (s)	SOAR (s)	DSR (s)
90	0.01	0.01	0.01	0.01	0.01	0.01	0.06	0.08	0.1
95	0.11	0.15	0.07	0.11	0.18	0.09	0.32	0.39	0.37
97	0.27	0.33	0.22	0.21	0.38	0.21	0.68	0.86	0.64
range	19.89	13.73	19.33	2.50	14.44	49.08	17.28	23.41	13.12

the partitioned network re-groups, it takes long time to re-discover the routes. The range for DSR (Table 5.4) for Mobile Relief Model is as high as 49 seconds. NEST also suffers from high range because it maintains on-demand routes to common nodes.

- There is an increase in delay when the netmark is more mobile.

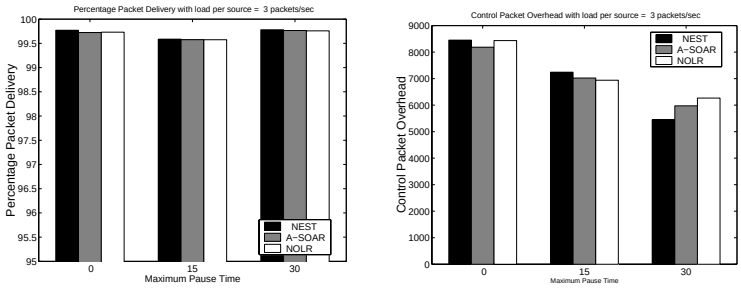
- NEST has better delay performance than SOAR because NEST has fewer queries and the paths tend to be more correct, so that less time is spent queueing packets at the routing layer or at the link layer. This indicates that the hybrid routing approach helps to reduce the end-to-end delay of the data packets.

5.4.6 Experimental Scenario 3

In this scenario there are two netmarks and 30 mobile nodes. The netmarks are placed at co-ordinates (250, 250) and (750, 250) and are fixed through out the simulation. The traffic pattern is according to the INTNET model. Packets from the Internet can enter the ad-hoc network through either of the two netmarks. Because

the netmarks advertise routes for the subnets and do not report the host routes, it can be safely assumed for the experiments that the incoming flows are randomly distributed between the two netmarks. Packets for the Internet are always forwarded to the nearest netmark. After a node decides which netmark to use, it encapsulates the IP packet with an IP header with destination being the netmark's address. In case of netmark-route unavailability, *anycast* queries (as discussed in Sec. 5.3) are sent and routes are established based on the *anycast* replies. We compare the performance of NEST with SOAR (denoted as anycast-enabled SOAR or A-SOAR in Fig. 5.9) and NOLR. Like A-SOAR, if necessary, NOLR and NEST also use *anycast* queries for netmarks.

From Fig. 5.9, we see that SOAR and NOLR both perform as well as NEST in terms of data delivery and control overhead under all three mobility scenarios. This is in contrast to the results presented in Sec. 5.4.4 and Sec. 5.4.5, where both SOAR and NOLR produce higher control overhead than NEST. This improvement of performance in SOAR can be attributed to the following three reasons: (a) If a route to a given netmark is not available, packets can still be sent to other netmarks, which helps in reducing the number of queries; (b) anycast route replies help to prevent flooding of queries and speed up the process of route discovery; and (c) reducing the number of query-reply packets prevents old link-state information from being injected into the network, which in effect reduces the number of updates. In our experiment involving multiple netmarks in a small network, these become dominant factors behind improving performance rather than node-centric routing. However, the proposed node-



(a) Percentage Data Delivery (b) Control Overhead

Figure 5.9: Performance of NEST, SOAR and NOLR for a network with 30 nodes and two netmarks

centric hybrid approach with proactive routes for netmarks is still attractive for this small network if the static or the hybrid method (Sec. 5.3) is used for forwarding outgoing packets because in those scenarios every node has to obtain path to every netmark.

5.5 Conclusions

We have presented node-centric approaches to hybrid routing for ad hoc networks that distinguish between normal nodes and special nodes called netmarks, which host popular network services or function as points of attachment to the Internet. With node-centric hybrid routing, netmarks force other common nodes to maintain routing information for them by either advertising their routing information as in table-driven routing protocols, or by requiring nodes to maintain routing entries towards them for extended periods of time. This reduces the network-wide flooding and the corresponding delay for route set up every time a session needs to be established between a normal node and a netmark. Routes between peer nodes are set up on-

demand. We have evaluated the changes needed to incorporate node-centric hybrid routing in the basic mechanism of routing for some pure on-demand routing protocols, namely AODV, DSR and SOAR and compared the performance of AODV, DSR and SOAR with the hybrid approaches, NEST and NOLR.

On the basis of ns2 simulations, we have found that, if a node in the ad hoc network acts as a source or relay of data packets for significant portion of its lifetime, the benefit of extending caching information in a purely on-demand routing protocol is not noticeable. However, maintaining proactive routes as in NEST offers better performance than any on-demand routing protocol, both in terms of data delivery and control packet overhead when the traffic flow is mostly from common nodes towards the netmark. We have also found that the performance of NEST is not affected by the mobility of netmarks. In a moderately-sized network served by multiple netmarks, the performance of on-demand routing protocols can be significantly improved by maintaining routes to any of the netmarks and then sending anycast queries asking for a route to the nearest netmark.

Chapter 6

Summary and Future Work

6.1 Contributions

The goal of this thesis has been to use link-state information on-demand for developing efficient routing protocols for mobile ad-hoc networks. The first protocol presented is source tree on-demand adaptive routing (SOAR) protocol that exchanges source-tree information, represented by links in an on-demand manner. This is the first routing protocol that uses link-state information in an on-demand manner. All the prior link-state routing protocols are proactive. Our study shows that SOAR is an effective protocol for mobile ad-hoc networks, and its approach to route reporting and route maintenance is more efficient than DSR and AODV, which are representative of the state-of-the-art on-demand routing protocols for mobile ad-hoc networks. The performance improvements obtained in SOAR compared to DSR and AODV is a direct result of communicating source trees rather than distances or paths to active

destinations in route requests and route replies. The source trees provide much more redundancy compared to other approaches and therefore, reduces the frequency with which network-wide route discoveries are needed. Moreover, SOAR can locally repair routes without sending route error notifications to the upstream nodes all the time, thereby minimizing the number of nodes affected due to network dynamics.

One of the novel design aspects for an on-demand link-state routing protocol is the path-selection algorithm. The constraint that the route through a neighbor is valid if it has been advertised by that neighbor makes the path selection problem a challenging one. We have at first presented a modified form of the Bellman-Ford algorithm that computes shortest valid routes for destinations, the combination of which forms a source graph. We have shown that the formation of a source tree based on constraints of on-demand link-state advertisements is an NP-complete problem and we have developed a polynomial-time approximating solution. Finally we have seen how the exchange of source tree combined with a path selection algorithm can be used for loop-free link-state policy based routing.

Building a source tree while satisfying the above path-selection constraint will make routes for several destinations infinite. Determining finite-cost routes for those destinations would involve extra control overhead if instantaneous loops must be prevented. Therefore, we developed the on-demand link-vector (OLIVE) protocol in which source graph is computed and the source graphs are advertised incrementally. In SOAR a source tree is advertised and source graph is computed locally. Instantaneous loop-freedom is ensured in OLIVE through synchronization with one-hop neighbors.

OLIVE does not need extra overhead in the data packets for detection of loops and the partial topology at any node provides sufficient information to prevent network-wide searches and repair routes locally.

We have introduced a novel node-centric hybrid routing approach for ad-hoc networks that distinguishes between common nodes and special nodes called netmarks, which host popular network services or function as points of attachment to the Internet. With node-centric hybrid routing, netmarks force other common nodes to maintain routing information for them by either advertising their routing information as in table-driven routing protocols, or by requiring nodes to maintain routing entries towards them for extended periods of time. This reduces network-wide flooding and the corresponding delay for route set up every time a session needs to be established between a normal node and a netmark. Routes between common nodes are set up on-demand. We have evaluated the changes needed to incorporate node-centric hybrid routing in the basic mechanism of routing for some pure on-demand routing protocols, namely AODV, DSR and SOAR and compared the performance of AODV, DSR and SOAR with the hybrid protocols NEST and NOLR. Simulation results have indicated node-centric hybrid routing approaches can significantly improve performance over pure on-demand routing protocols.

This dissertation includes materials that have been previously published in [38, 42, 40, 39, 41]. Co-author, JJ Garcia-Luna-Aceves listed directed and supervised the research that forms the basis of this dissertation.

6.2 Future Work

One important piece of future work is to develop a hierarchical version of OLIVE that can scale to a large network. Moreover, it would be interesting to determine how much improvement in performance can a node-centric hybrid version of OLIVE provide.

Most of the simulations done to date concentrate on networks with an average hop count smaller than two and a high node-density. Few research results are available for finding the impact of node-density and path-length on the performance of the routing protocol. One reason for this is the absence of a mobility model that can be parameterized by node density and path length. Therefore, one challenging future research work would be to develop such a mobility model and compare the performance of DSR, AODV, OLIVE, TBRPF and OLSR by varying node-density and average path length.

When networks become large, flood-search messages become very expensive making localized search highly preferable. Solutions proposed in literature to prevent complete flooding focus on dividing the network into clusters in which proactive routing is used in clusters and reactive routing happens between clusters.

We seek a solution that is simple and practical and does not have the complexity of hierarchical address maintenance. A possible solution consists of dividing the network into clusters without the added complexity of cluster formation and maintenance. In a large network. when there are multiple netmarks each mobile node based on any metric can select any of the netmarks as the "closest" netmark. If we group the

nodes based on their "closest" netmark, then it will automatically lead to the formation of non-overlapping clusters (see Fig. 6.1), without the extra overhead of explicit cluster formation and maintenance.

Assume A, B, C, D are four netmarks which are serving the mobile ad-hoc network. When common nodes affiliate themselves with the nearest netmarks, clusters CA, CB, CC and CD get formed automatically. The basic idea is that each mobile node knows the nearest netmark, and when it changes affiliation from one netmark to the other, it sends an update to the old and the new primary netmark, informing them of their new affiliations. Accordingly, each netmark knows which mobile nodes are affiliated to it. Every node maintains a sequence number for the affiliation message it sends to a netmark, so that in case of collisions, conflicts can be resolved easily. When a node s wants to communicate with another mobile node d (Fig. 6.1), if it does not have a route for the destination, it asks each netmark whether node d is affiliated to any one of them. When a netmark (D) responds that node d is affiliated with netmark D, node s will send a query towards that netmark. The first node x within Cluster CD will initiate a flood-query search that will be limited among nodes affiliated with D i.e. in cluster CD. The destination or the intermediate node who has a path to node d will respond to node x who replies back to node s. It is interesting to note that node x automatically starts working as a boundary router without the added complexity of selecting the boundary routers.

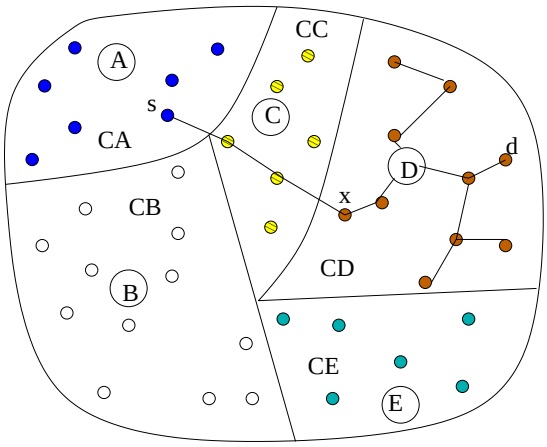


Figure 6.1: Cluster formation in multi-netmark networks

Bibliography

[1] R. Albrightson, J. J. Garcia-Luna-Aceves, and J. Boyle. EIGRP-A Fast Routing Protocol Based on Distance Vectors. In *Proc. Networld/Interop 94*, Las Vegas, Nevada, June, 1988.

[2] C.R. Baucgh, J. Huang, R. Schwartz, and D. Trinkwon. Traffic Model for 802.16 TG3 MAC/PHY SImulations. In *http://ieee802.org/16*, 2001.

[3] J. Behrens and J. J. Garcia-Luna-Aceves. Distributed, Scalable Routing Based on Link-State Vectors. In *ACM Special Interest Group on Data Communication (Sigcomm)*, pages 136–147, 1994.

[4] T. Clausen. OLSR ns2 simulation code . In *http://hipercom.inria.fr/olsr/*. INRIA, October 18 2000.

[5] IEEE Computer Society LAN MAN Standards Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. The Institute of Electrical and Electronics Engineers, 1997. IEEE Std 802.11.

[6] T. H. Cormen, C. H. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice-Hall, 1999.

[7] K. Leibnitz D. Staehle and P. T. Gia. Source Traffic Modeling of Wireless Applications. In *CNO Activity Report 1999/2000*, June, 2000.

[8] S. R. Das, C. E. Perkins, and E. M. Royer. Performance Comparison of Two On-Demand Routing Protocols for Ad-Hoc Networks. In *Proc. IEEE Conference on Computer Communications (Infocom)*, pages 3–12, Tel Aviv, Israel, March 26 - 30 2000.

[9] J. Broch et. al. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proc. ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 85–97, Dallas, Texas, October 25-30 1998.

[10] P. Johansson et. al. Scenario Based Performance Analysis of Routing Protocols for Mobile Ad-Hoc Networks. In *Proc. ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 195–206, Seattle, Washington, August 15-20 1999.

[11] L.M. Feeney. Investigating the Energy-consumption Model of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proc. IEEE Conference on Computer Communications (Infocom)*, pages 1548–1557, Anchorage, Alaska, April 22-26 2001.

[12] E. Gafni and D. Bertsekas. Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology. *IEEE Transactions on Communications*, 29(1):11–15, 1981.

[13] M. Garey and D. Johnson. *Computers and Intractability. A Guide to the theory of NP-completeness.* W. H. Freeman and Company, 1979.

[14] T. Griffin and G. T. Wilfong. A Safe Path Vector Protocol. In *Proc. IEEE Conference on Computer Communications (Infocom)*, pages 490–499, Tel-Aviv, Israel, March 22-26 2000.

[15] Z. Haas and M. R. Pearlman. The Zone Routing Protocol (ZRP) for Ad Hoc Networks. In *http://www.ee.cornell.edu/ haas/Publications/draft-ietf-manet-zone-zrp-02.txt*, June 1999.

[16] C. Hedrick. Routing Information Protocol. In *RFC 1058*, June, 1988.

[17] Mobile Ad hoc Networks (manet). http://www.ietf.org/html.charters/manet-charter.html.

[18] http://www.isi.edu/nsnam/ns/. The Network Simulator - ns-2. ns-2.1b6.

[19] http://www.isi.edu/nsnam/ns/. The Network Simulator - ns-2. ns-2.1b8.

[20] Y. C. Hu and D. Johnson. Implicit Source Routes for On-Demand Ad Hoc Network Routing. In *Proc. ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 1–10, Long Beach, California, October 4-5 2001.

[21] Y.C. Hu and David Johnson. Caching Strategies in On-Demand Routing Protocols for Wireless Ad Hoc Networks. In *Proc. ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 231–242, Boston, Massachusetts, August 6-11 2000.

[22] P. Jacquet and et al. Optimized Link State Routing Protocol. In *http://hipercom.inria.fr/olsr/draft-ietf-manet-olsr-09.txt*, April 15 2003.

[23] H. Jiang and J.J. Garcia-Luna-Aceves. Performance Comparison of Three Routing protocols for Ad Hoc networks. In *Proc. Twelfth International Conference on Computer Communications and Networks (ICCCN)*, Phoenix, Arizona, October 15-17 2001.

[24] J.J.Garcia-Luna-Aceves and M.Spohn. Transmission-Efficient Routing in Wireless Networks using Link-State Information. *ACM Mobile Networks and Applications Journal*, 6(3):223–238, 2001.

[25] D. B. Johnson and D. A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[26] J. J. Garcia Luna-Aceves and M. Spohn. Scalable Link-State Internet Routing. In *Proc. IEEE International Conference on Network Protocols (ICNP)*, pages 52–61, Austin, Texas, October 13-16 1998.

[27] M. Marina. AODV code for CMU Wireless and Mobility Extensions to ns-2 . In *http://www.ececs.uc.edu/ mmarina/aodv/*, last updated on 12/07/2000.

[28] J. Moy. *OSPF Version 2*, July 1991. RFC 1247.

[29] R. Ogier. A Simulation Comparison of TBRPF, OLSR, and AODV. In *http://www.erg.sri.com/projects/tbrpf/*, July 2002.

[30] R. G. Ogier, F. L. Templin, B. Bellur, and M. G. Lewis. Topology Broadcast Based on Reverse-Path Forwarding (TBRPF). In *http://www.potaroo.net/ietf/ids/draft-ietf-manet-tbrpf-08.txt*, April 22 2003. Internet Draft.

[31] International Standards Organization. Intra-Domain IS-IS Routing Protocol. In *ISO/IEC JTC1/SC6 WG2 N323*, Sep, 1989.

[32] V. D. Park and M. S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proc. IEEE Conference on Computer Communications (Infocom)*, pages 1405–1413, Kobe, Japan, April 7-12 1997.

[33] C. E. Perkins and E. M. Royer. Ad Hoc On-Demand Distance Vector Routing. In *Proc. of IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100, New Orleans, LA, February 25-26 1999.

[34] C. E. Perkins, E. M. Royer, and S. R.Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. In *http://www.potaroo.net/ietf/ids/draft-ietf-manet-aodv-13.txt*. Mobile Ad Hoc Networking Working Group, February 13 2003. Internet Draft.

[35] J. Raju and J. J. Garcia-Luna-Aceves. Efficient On-Demand Routing Using

Source-Tracing in Wireless Networks. In *Proc. IEEE Global Telecommunications Conference (Globecom)*, San Francisco, CA, November 27-30 2000.

[36] J. Raju and J.J. Garcia-Luna-Aceves. A New Approach to On-Demand Loop-Free Multipath Routing. In *Proc. IEEE Twelfth International Conference on Computer Communications and Networks (ICCCN)*, pages 522–527, Boston, Massachusetts, October 11-13 1999.

[37] Y. Rekhter and T. Li. A Border Gateway Protocol (BGP-4). In *RFC 1771 (BGP version 4)*, 1995.

[38] S. Roy and J.J. Garcia Luna Aceves. Using Minimal Source Trees for On-Demand Routing in Ad Hoc Networks . In *Proc. IEEE Conference on Computer Communications (Infocom)*, pages 1172–1181, Anchorage, Alaska, April 22-26 2001.

[39] S. Roy and J. J. Garcia-Luna-Aceves. Node-Centric Hybrid Routing for Ad Hoc Networks. In *10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- Workshops (MASCOTS 2002 Workshops)*, Fort Worth, Texas, October 12-16 2002.

[40] S. Roy and J. J. Garcia-Luna-Aceves. Node-Centric Hybrid Routing for Ad Hoc Wireless Extensions of The Internet. In *Proc. IEEE Global Telecommunications Conference (Globecom)*, Taipei, Taiwan, November 17-21 2002.

[41] S. Roy and J. J. Garcia-Luna-Aceves. Node-Centric Hybrid Routing for Wireless Internetworking. In K. Makki, editor, *Mobile and Wireless Internet*, pages 191–216 (Chapter 7). Kluwer Academic Publishers, 2002.

[42] S. Roy and J.J. Garcia-Luna-Aceves. An Efficient Path Selection Algorithm for On-Demand Link-State Hop-by-Hop Routing. In *Proc. IEEE International Conference on Computer Communications and Networks (ICCCN)*, Miami, Florida, October 14-16 2002.

[43] E. M. Royer, Y. Sun, and C. Perkins. Global Connectivity for IPv4 Mobile Ad hoc Networks. In *www.cs.ucsb.edu/ ebelding/txt/globalv4.txt*. Mobile Ad Hoc Networking Working Group, November 14 2001. Internet Draft.

[44] M. Spohn and J. J. Garcia-Luna-Aceves. Neighborhood Aware Source Routing. In *Proc. ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 11–21, Long Beach, California, October 4-5 2001.

[45] P. F. Tsuchiya. The Landmark Hierarchy: a New Hierarchy for Routing in very Large Networks. In *ACM Special Interest Group on Data Communication (Sigcomm)*, pages 35–42, Stanford, CA, August 16-18 1988.

[46] B. Tuch. Development of WaveLAN, an ISM band wireless LAN. *AT&T Technical Journal*, 72(4):27–33, July/Aug 1993.

[47] K. Varadhan, R. Govindan, and D. Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks (Amsterdam, Netherlands: 1999)*, 32(1):1–16, 2000.

[48] W.T. Zaumen and J.J. Garcia-Luna-Aceves. Dynamics of Distributed Shortest-Path Routing Algorithms. In *ACM Special Interest Group on Data Communication (Sigcomm)*, pages 31–43, Zurich, Switzerland, September 3-6 1991.

[49] Z. Zhan. One in Three SAT (1in3SAT) Problem. In *www.eecs.wsu.edu/ cs516/notes34.ps*, 2000.